TU Clausthal

Edward Fischer

Operationalisierung des Projektcontrollings

SSE-Dissertation 2

Software
Systems
Engineering

Institut für Informatik
Lehrstuhl von Prof. Dr. Andreas Rausch

Operationalisierung des Projektcontrollings

D i s s e r t a t i o n

zur Erlangung des Grades eines Doktors
der Naturwissenschaften

vorgelegt von
Edward Fischer
aus Dresden

genehmigt von der Fakultät für
Mathematik/Informatik und Maschinenbau
der Technischen Universität Clausthal

Tag der mündlichen Prüfung
24.09.2010

Vorsitzender der Promotionskommission:
Prof. Dr. Garbiel Zachmann
Hauptberichterstatter:
Prof. Dr. Andreas Rausch
Berichterstatter:
Prof. Dr. Rüdiger Liskowsky

D 104
Dissertation Clausthal 2010

Kurzfassung

Der Verlauf von Projekten ist schwer vorherzusagen – besonders bei der Entwicklung von Systemen der Informationstechnologie. Bisherige Erfahrungen können wegen verschiedenster Einflussfaktoren nur bedingt auf neue Situationen übertragen werden. Doch je detaillierter das geplante Soll und das aktuell erreichte Ist durch ein Projektcontrolling verfolgt wird, desto eher ist es möglich, Probleme zu erkennen und darauf steuernd zu reagieren.

Für einen hohen Detailgrad des Projektcontrollings ist eine Werkzeugunterstützung erforderlich. Dafür ist der informal definierte Begriff des Projektcontrollings mit formalen, automatisiert ausführbaren Operationen zu unterlegen. Eine solche Operationalisierung kann sich dabei nicht auf die Generierung eines Solls – z.B. anhand eines Vorgehensmodells – beschränken, sondern muss im Kontext manueller Tätigkeiten betrachtet werden: Das Soll wird um Planungsdaten wie Termine und Ressourcen angereichert. Auch wird diesem – im Sinne der Fortschrittskontrolle – das bisher Erreichte zugeordnet. Diese Zuordnungen dienen wiederum als Eingabe zur Ableitung des jeweils nächsten Solls, sofern die Inhalte des bisher Erreichten für die weitere Planung ausgewertet werden sollen.

Diese Arbeit stellt eine Lösung zur Operationalisierung des Projektcontrollings vor, die die automatisierte Berechnung des Solls mit den manuellen Tätigkeiten der Planungsausgestaltung und Fortschrittskontrolle ohne Informationsverlust integriert. Eine solche Berechnung wird auch als eine inkrementelle Transformation bezeichnet. Grundlage ist ein formales Modell für Projekte die anhand eines Vorgehensmodells abgewickelt werden. Die Vorgaben des Vorgehensmodells werden durch eine inkrementelle Transformation auf ein Projekt angewendet, wodurch das Soll nur aktualisiert wird, anstelle es stets von Grund auf neu zu erstellen. Da das Vorgehensmodell Teil des Projektes ist, können die Vorgaben auf sich selbst bzw. auf andere Vorgaben beziehen, so dass auch die Anpassung eines Vorgehensmodells für die Steuerung in Teilen automatisiert werden kann. Eine Automatisierung der Fortschrittskontrolle ist dagegen im Allgemeinen nicht entscheidbar. Denn in der Durchführung können Fehler entstehen, deren Auflösung die Auswertung nicht formal greifbarer Hergänge erfordern kann. Daher erfolgt hier nur eine Teilautomatisierung durch Berechnung von Vorschlägen zur Zuordnung der Elemente des Ists zu denen des Solls.

Um Vorgaben eines Vorgehensmodells als inkrementelle Transformation anwenden zu können, müssen diese formal und in Form von Graphersetzungsregeln vorliegen. Dies ist bei heutigen Vorgehensmodellen einerseits nicht der Fall, und andererseits nicht zu erwarten, da deren Autoren sich eher auf einer fachlichen als auf einer technischen Ebene bewegen. Als Brücke umfasst die Lösung dieser Arbeit eine Integrationsmethodik, anhand der die für Vorgehensmodelle in der Praxis tatsächlich eingesetzten Sprachen mit der Automatisierung der Soll-Berechnung integriert werden können. Grundlage dieser Methodik ist die Betrachtung einer Sprache als eine Benutzungsschnittstelle, die wie in einem Projekt zu entwickeln ist. Konkret werden dazu UML-Aktivitätsdiagramme eingesetzt, um die möglichen Vorgaben durch strukturierte Texte zu beschreiben.

Eine abschließende Betrachtung zeigt, dass die Lösung leicht in die existierende Werkzeuglandschaft – mit der bisher die manuelle Planung, Fortschrittskontrolle und Definition von Vorgehensmodellen erfolgt – eingegliedert werden kann.

Danksagung

Mit dem V-Modell durch Deutschland und die Welt – so kann der Auftakt und der Rahmen dieser Arbeit rückblickend zusammengefasst werden. Die Mitarbeit an den vielen, sehr interessanten Projekten rund um das V-Modell am Lehrstuhl von Prof. Dr. Andreas Rausch ermöglichte es, bei theoretischen Gedankengänden auch stets den Bezug zur Realität zu wahren. In diesem Zusammenhang gebührt mein Dank an Prof. Dr. Andreas Rausch für die mir zur Promotion umfangreich eingeräumten Zeit, sowie die seinige für zahl- und hilfreiche Diskussionen. Durch deren Reibung konnte diese Arbeit nach und nach verbessert werden. Bedanken möchte ich mich aber auch für die äußerst angenehme Atmosphäre, die Prof. Dr. Andreas Rausch und meine Kollegen am Lehrstuhl aufgebaut haben, die durch höchste Integrität, Hilfsbereitschaft und Unterhaltungswert, gekennzeichnet sind. Sei es die Arbeit an Publikationen (Danke, Andreas und all die anderen!), logistische und kulinarische Unterstützung (Danke, Dirk und Sabine Niebuhr!) oder die Kicker-Huldigungen, Küchen- oder sonstige Diskussionen (Danke, Christian Bartelt, Dirk und Thomas Bravin!).

Sowohl für die Übernahme des Zweitgutachtens, als auch für den Auftakt der Promotion am Ende meines Studiums, gebührt mein Dank Prof. Dr. Rüdiger Liskowsky. Dies schließt auch den Großen Beleg und die Diplomarbeit mit ein, mit der die V-Modell-Reise ihren Ursprung fand. Auch hier möchte ich mich für den Freiraum noch einmal herzlichst bedanken.

Für die Unterstützung in der Anfangs- und Endphase möchte ich mich auch bei Dr. Jan Zöllner bzw. Thomas Ternité bedanken. Nicht zu letzt gilt mein Dank auch meinem Vater für die Prägung meines Bildungswegs – besonders für die Wahl des Martin-Andersen-Nexö-Gymnasiums, mit der mein Interesse an der Informatik begann.

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

Festpreis oder pro Stunde? Diese Frage ist besonders für Projekte interessant, da sie darüber entscheidet, wer das finanzielle Risiko trägt. Ein Auftraggeber wird also eher die erste, ein Auftragnehmer eher die zweite Abrechnungsart bevorzugen. Doch wie kommt dieses Risiko überhaupt zustande? Warum ist der letztendliche Preis vor, und selbst noch während eines Projektes, nicht so genau bestimmbar? Zunächst bedingt die Einmaligkeit von Projekten, dass Erfahrungswerte ähnlicher Vorhaben möglicherweise nicht übertragbar sind: Schon allein die Fähigkeiten der eingesetzten Personen oder geringfügig abweichende Ziele, die sich gegebenenfalls auch noch erst während des Projektes konkretisieren, können neue Probleme aufbringen die einen ganz anderen Verlauf nach sich ziehen. Eine zweite Ursache besteht in der Komplexität heutiger Projekte, insbesondere wenn diese mit der immer mehr Bereiche des Lebens durchdringenden Informationstechnologie zu tun haben: Während bei einem traditionellen, „physikalischen" System Wechselwirkungskräfte mit steigendem Abstand sinken, kann bei einem Softwaresystem eine beliebig kleine „Kraft" eine beliebig „weit entfernte" Wirkung erzielen. Dies führt dazu, dass potenziell alle Teile eines Systems miteinander in Beziehung stehen, und damit die zu berücksichtigenden Wechselwirkungen deutlich schwieriger zu überblicken oder gar abzuschätzen sind. Dies gilt umso mehr, je umfangreicher das entwickelte System oder die Zahl der beteiligten Personen ist.

Wie gehen wir im Projekt vor? Trotz aller Ungewissheiten ist ein planvolles Vorgehen sinnvoll: Je feiner ein Projekt in kleine Teile zerlegt wird, desto genauer kann auch eine Schätzung und Kontrolle des Fortschritts erfolgen. Zudem wird es klarer, ob Abweichungen vom Soll tatsächlich einer schlechten handwerklichen Durchführung, oder nicht eher einer unrealistischen Zielvorgabe zuzurechnen sind. Da dies bei einer feingliedrigen Planung schon früh im Projekt erkennbar ist, kann ein Projektleiter noch steuernd eingreifen. Dieser Kreislauf aus Planung, Durchführung, Kontrolle und Steuerung wird als Projektcontrolling bezeichnet. Um bereits beim Aufstellen eines Projektplans Wesentliches nicht zu übersehen, helfen Vorgehensmodelle. Dazu enthalten sie meist informale Vorgaben, die beschreiben, welche Teilaufgaben initial, und welche weiteren anhand des jeweils Erarbeiteten einzuplanen sind. Die Anwendung dieser Vorgaben stößt bei einem manuellen Projektcontrolling bei größeren Projekten in der Praxis spätestens auf Ebene von Dokumenten an die Grenze des machbaren, obwohl auch deren Inhalte und Versionen relevante Größen sind [1,2]. Es liegt daher nahe, die mit Prozessbeschreibungssprachen wie SPEM [3] überwiegend informal beschriebenen Vorgaben mit konkret und automatisiert ausführbaren Operationen für die einzelnen Schritte des Projektcontrollings zu unterlegen.

Workflowsysteme als Operationalisierung? Workflowsysteme (wie etwa ClearQuest [4] oder Notes [5]) setzen konzeptionell (vgl. [6]) eine feste Reihenfolge der abzuarbeitenden Teilaufgaben voraus – der aktuelle Projektplan entspricht dabei der Menge der gerade anstehenden Teilaufgaben. Doch je detaillierter ein Projekt zerlegt wird, desto unwahrscheinlicher ist es, dass dessen Teilergebnisse auch genau in der vorher definierten Reihenfolge erstellt werden können. Einmal gestartete Workflowabläufe könnten zwar durch Mechanismen wie in [7] abgebrochen, und von einem früheren Punkt aus wieder aufgenommen werden. Doch wenn nach einem solchen Rücksprung nicht das gesamte Projekt anders verläuft, gehen die zu den noch weiterhin anstehenden Teilaufgaben zwischenzeitlich zugeordneten Planungsdaten (wie Termine oder Ressourcen) verloren. Zudem würde ein auf Workflows basierender Projektplan stets nur die unmittelbar als nächstes anstehenden Teilaufgaben enthalten – und nicht etwa auch solche, die absehbar später noch kommen werden. Dadurch wiederum sinkt die Genauigkeit der darauf beruhenden Schätzungen. Zwar

bieten Workflowsysteme die Möglichkeit (vgl. [8]), beliebige Algorithmen zum Umgang mit Änderungen und der vorausschauenden Planung zu integrieren – doch müssten diese vom Anwender erst erdacht und programmiert werden. Es stellt sich also die Frage, ob dazu bereits existierende, möglichweise allgemeinere Konzepte wiederverwendet werden können.

Umweg über Versionsverwaltungssysteme? Ein vielversprechendes Konzept zum Umgang mit Änderungen sind Versionsverwaltungssysteme wie CVS [9] oder Subversion [10]. Diese können genutzt werden, um parallel an einem Dokument zu arbeiten. Ausgehend von einem gemeinsamen Stand können mehre Personen durch das Beziehen einer Kopie und deren anschließender Bearbeitung unterschiedliche eigene Varianten ableiten. Diese Varianten können später zusammengeführt werden, so dass wieder ein gemeinsamer Stand erreicht ist, der alle Änderungen aller Personen vereint. Zur Anwendung für die Operationalisierung des Projektcontrollings könnte die eine Person der Projektleiter sein, der den Projektplan manuell um Planungsdaten erweitert, während die andere Person durch ein Programm ersetzt werden würde, welches stets einen neuen Projektplan anhand des aktuellen Ist generiert. So könnte der Zusammenführungsmechanismus der Versionsverwaltungssysteme genutzt werden, um den Projektplan zu aktualisieren. Das funktioniert jedoch nicht, da der zugrunde liegende Mechanismus, RCS [11], die Dokumente über einen textbasierten Zeilenvergleich zusammenführt. Konkret wird weder sichergestellt, dass bei mehreren gleichen bzw. ähnlichen Zeilen die Übereinstimmungen korrekt erkannt werden, noch dass überhaupt ein strukturell korrekter Projektplan herauskommt (vgl. [12]). Somit muss der Anwender nach jeder Zusammenführung Konflikte auflösen und sogar noch auf vom Verfahren nicht automatisch erkannte Fehler bei falschen Übereinstimmungen prüfen.

Modellbasierte Versionsverwaltung? Die Modellbasierte Versionsverwaltung ist eine strukturelle Verfeinerung von RCS. Anstelle von Textzeilen werden Graphen verwendet um die Inhalte eines Dokumentes zu repräsentieren. Zum Feststellen der Übereinstimmungen der Inhalte zweier Varianten eines Dokumentes werden zwei prinzipielle Ansätze verfolgt: Strukturelle Ähnlichkeit (die durch die Graphen feingranularer als bei Textzeilen sein kann) [13,14] und UUIDs (Universally Unique Identifiers) [15,16]. Anders als bei Modellen gibt es in Projektplänen eine Vielzahl von sehr ähnlichen Einträgen, besonders wenn diese generiert werden. Damit bleibt beim strukturellen Vergleich, egal ob über Zeilen oder Graphen, eine hohe Wahrscheinlichkeit einer Fehlzuordnung. Die Generierung des Projektplans ist auch der Grund, warum der Einsatz von UUIDs nicht weiterhilft: Per Definition erzeugt jede Anwendung der Generierung ein Projektplan dessen Elemente stets neue UUIDs besitzen.

Inkrementelle Transformation! Die inkrementelle Transformation [17,18] ist ein Algorithmus, der Berechnungsvorschriften (wie die zum Generieren eines Projektplans) aktualisierend ausführen kann. Zur Operationalisierung des Projektcontrollings müsste ein Anwender somit den Umgang mit Änderungen nicht selbst implementieren, sondern könnte sich auf die Fachlichkeit der Vorgaben des betrachteten Vorgehensmodells konzentrieren. Entsprechende Ansätze wie in [19-22] setzen jedoch voraus, dass diese Vorgaben mit Graphersetzungsregeln beschrieben werden. Diese wiederum befinden sich auf einem zu niedrigen Abstraktionsniveau, um für die Praxis anwendbar zu sein [23]. Das Hauptproblem bilden dabei komplexe Bedingungen, wie beispielsweise die Forderung, im Projekt pro Iteration nur die jeweils aktuellste fertig gestellte Version einer Systemarchitektur zu betrachten, um für jedes darin gegenüber der bisherigen Iteration neu identifizierte Systemelement eine Spezifikation einzuplanen. Somit existiert auch auch mit diesem Ansatz noch keine brauchbare Lösung.

1.2 Ziele und Ergebnisse

Ziel dieser Arbeit ist es, die für einen Projektleiter zentralen Schritte des Projektcontrollings, also die Planung, die Kontrolle und die Steuerung, zu operationalisieren, d.h. zu automatisieren oder durch Teilautomatisierungen zumindest zu unterstützen. Für den Schritt der Planung umfasst dies die initiale Erstellung, sowie die laufende Aktualisierung, eines Projektplans (allgemein des Solls) gemäß den Vorgaben eines Vorgehensmodells. Während die initiale Erstellung einzig auf den Vorgaben beruht, müssen für die Aktualisierung zusätzlich noch die jeweils bis dahin erarbeiteten Dokumente etc. (allgemein das Ist) ausgewertet werden. Das wiederum erfordert es, das aktuelle Ist zu dem jeweils bisherigen Soll zuzuordnen, da sonst nicht klar ist, welche Dokumente bzw. welche Versionen für die Aktualisierung (also für die Berechnung des neuen Solls) heranzuziehen sind. Diese Zuordnung entspricht dem Schritt der Kontrolle. Das vom Projektteam erarbeitete Ist kann Fehler bezüglich des Solls aufweisen. Bei der Kontrolle gilt es diese aufzudecken, um eine korrekte Zuordnung sicherzustellen. Das Aufdecken kann im Allgemeinen jedoch nicht automatisiert werden, da Fehler auch auf nicht formal greifbaren Umständen beruhen können. Für die Zuordnung wird daher nur eine Teilautomatisierung durch Berechnung von Zuordnungsvorschlägen verfolgt, jene vom Projektleiter anschließend manuell zu den tatsächlich gültigen Zuordnungen zu überführen sind. Aus der Kontrolle ergeben sich auch die Konsequenzen für den weiteren Ablauf im Projekt: Im einfachsten Fall ist mit der Durchführung fortzufahren. Andernfalls muss das Soll in einem weiteren Planungsschritt aktualisiert, oder sogar das Vorgehensmodell angepasst werden. Letzteres entspricht dem Schritt der Steuerung. Für die Automatisierung wird dazu nur die Auswahl der für die aktuelle Projektsituation relevanten Vorgaben eines Vorgehensmodells, anhand des bisher erarbeiteten Ist, verfolgt.

Die Lösung besteht darin, die inkrementelle Transformation mit Prozessbeschreibungssprachen zu integrieren. So können Vorgehensmodelle auf einem hohen Abstraktionsniveau definiert werden, wobei deren Vorgaben dennoch automatisiert und inkrementell ausgewertet werden können. Aus diesem Ansatz ergeben sich die folgenden wesentlichen Ergebnisse dieser Arbeit:

- **Grundlagenerweiterungen.** Die Grundlagenerweiterungen bauen die inkrementelle Transformation um weitere Konzepte aus, wodurch die eigentliche Lösung überhaupt erst vertretbar dargestellt werden kann.
- **Automatisierungskonzept.** Das Automatisierungskonzept beschreibt, wie die inkrementelle Transformation für das Projektcontrolling angewendet werden kann.
- **Integrationsmethodik.** Die Integrationsmethodik beschreibt, wie eine existierende Prozessbeschreibungssprache aus der Praxis mit dem Automatisierungskonzept integriert werden kann. Dadurch ist die Lösung nicht auf eine konkrete Prozessbeschreibungssprache begrenzt, sondern kann auch allgemein angewendet werden.

Die Ergebnisse dieser Arbeit sind für folgende Zielgruppen interessant:

- **Projektleiter.** Eine Implementierung der Lösung vorausgesetzt, erhalten Projektleiter (Teil-)Automatisierungen für die Planung, Kontrolle und Steuerung von Projekten, mit denen sie ein sowohl detailliertes als auch aktuelles Projektcontrolling durchführen können. Dies erlaubt es ihnen, genauere Prognosen zum Projektverlauf und den benötigten Ressourcen aufzustellen, was wiederum z.B. finanzielle Projektrisiken mindert.
- **Prozessingenieure.** Prozessingenieure erhalten mit dem Integrationskonzept eine Möglichkeit, Prozessbeschreibungssprachen zu definieren die einerseits dem bisherigen, für solche Sprachen üblichen Abstraktionsniveau entsprechen, andererseits

die damit beschriebenen Vorgehensmodelle dennoch für ein automatisiertes Projektcontrolling auswertbar sind.

- **Werkzeughersteller.** Die aufgestellte Lösung ist für Werkzeughersteller insofern interessant, als dass sie erst durch ihre Implementierung den vorherigen beiden Zielgruppen zugänglich wird.

Gemäß den Zielgruppen werden folgende Ergebnisse zur Erprobung und Veranschaulichung der Lösung erarbeitet:

- **Prozessbeschreibungssprache PBS$^+$.** Die Prozessbeschreibungssprache PBS$^+$. ist ein um das Automatisierungskonzept erweiterter Ausschnitt der Prozessbeschreibungssprache V-Modell XT 1.3 Metamodell, und ist daher von einem Prozessingenieur auf dem gleichen Abstraktionsniveau nutzbar.
- **Vorgehensmodell VM$^+$.** Das in PBS$^+$ beschriebenen Vorgehensmodell VM$^+$ dient Prozessingenieuren (und Projektleitern die während des Projektes in diese Rolle schlüpfen) als Fallbeispiel zur Definition eigener Vorgehensmodelle.
- **Integrationsansatz für das Werkzeugumfeld des V-Modell XT.** Anhand der im Umfeld des V-Modell XT verwendeten Werkzeuge wird gezeigt, dass und wie die Lösung in der Praxis eingesetzt werden kann – sowohl zur Definition von Vorgehensmodellen, als auch zu deren Anwendung

1.3 Inhalt und Aufbau

Nachfolgend werden der Inhalt und der logische Aufbau der Arbeit skizziert.

Grundlagen und Stand der Technik. Die Behandlung der Grundlagen zur Operationalisierung und zum Projektcontrolling erstreckten sich über zwei aufeinander aufbauende Kapitel.

Zunächst werden die für formale Definitionen verwendeten algebraischen Mittel fixiert. Nach Klärung der zentralen Begriffe formaler Sprachen werden darauf aufbauend deren Beschreibung durch Graphgrammatiken und (Meta-)Modellen behandelt. Damit geht die Betrachtung von Transformationen und der Möglichkeiten einher, diese inkrementell zu gestalten, d.h. das Ergebnis bei mehreren Ausführungen in Folge nur bezüglich der notwendigen Änderungen zu aktualisieren ohne es jedes Mal komplett neu zu erstellen.

Anschließend wird der Bezug dieser allgemein anwendbaren Konzepte zu Vorgehensmodellen (als Sprachen für Projekte bzw. Projektstände) und Prozessbeschreibungssprachen (als Sprachen für Vorgehensmodelle) hergestellt, und die entsprechenden Begriffe festgezurrt. Auch werden konkrete Prozessbeschreibungssprachen hinsichtlich ihrer Möglichkeiten und Defizite bezüglich der inkrementellen Transformation behandelt. Um die Problemstellung dieser Arbeit und dessen Lösung anschaulich darzustellen, wird in diesem Kapitel ein konkretes Beispiel eines Vorgehensmodells (das VM_0) definiert.

Problemstellung. In diesem Kapitel wird der Begriff des Projektcontrollings formal definiert und anhand des Vorgehensmodellbeispiels erläutert. Es wird gezeigt, warum die Operationalisierung des Projektcontrollings sowohl für den Projektleiter als auch für den Prozessingenieur ein zusammenhängendes Problem darstellt, und welche Aufgabenstellung sich diese Arbeit diesbezüglich setzt.

Lösung. In diesem Kapitel wird die Lösung der Problemstellung in ihren wesentlichen Punkten kompakt dargestellt. Insbesondere werden dabei die Herausforderungen bei der Anwendung der inkrementellen Transformation zur Beschreibung eines Vorgehensmodells erörtert und deren Bewältigung aufgezeigt. Die Lösung gliedert sich in eine Erweiterung der

Grundlagen, das Automatisierungskonzept sowie eine diesbezügliche Integrationsmethodik für Prozessbeschreibungssprachen.

Evaluation. In diesem Kapitel wird die Lösung anhand des durchgängigen Beispiels aus der Praxis evaluiert: Als erster Schritt wird die für VM_0 verwendete Prozessbeschreibungssprache (das V-Modell XT Metamodell in der Version 1.3) bezüglich des für VM_0 relevanten Ausschnitts und anhand der Integrationsmethodik mit dem Automatisierungskonzept integriert. Es entsteht die Prozessbeschreibungssprache PBS^+, die in Syntax und Semantik vorliegt. Als zweiter Schritt wird VM_0 auf PBS^+ migriert, wodurch VM^+ entsteht. Die Migration besteht darin, die in VM_0 noch informalen Vorgaben durch die mit PBS^+ bereitgestellten Sprachkonstrukte zu formalisieren. Anschließend wird gezeigt, wie die formalisierten Vorgaben im Rahmen des Automatisierungskonzepts angewendet werden, um Projektpläne ohne Informationsverlust zu aktualisieren. Abschließend wird aufgezeigt, dass und wie die Lösung auch mit der in der Praxis vorherrschenden Werkzeugumgebung integrierbar ist.

2 Grundlagen zur Operationalisierung

Unter Operationalisierung[1] sei die Unterlegung einer nicht formal definierten Tätigkeit (wie z.B. des Projektcontrollings) mit konkret und automatisiert ausführbaren Operationen verstanden. In diesem Kapitel werden die dafür notwendigen Begriffe und Notationen (kurz: Konzepte) eingeführt.

2.1 Mathematische Grundkonzepte

Folgende Konzepte werden als bekannt vorausgesetzt:

- mathematisches Objekt, Menge, Teilmenge, Element (vgl. [25])
- Menge der natürlichen Zahlen, notiert mit $\mathbb{N}$
- Kardinalität (vgl. [25]) einer Menge M, notiert mit | M |
- Potenzmenge (vgl. [25]) einer Menge M, mit Pot(M)
- Tupel, Stelligkeit, Projektion, Relation (vgl. [26])
- Kartesisches Produkt (vgl. [26]) zweier Mengen M und N, notiert mit $M \times N$. Falls M=N, kann abkürzend auch M^2 (oder N^2) geschrieben werden.
- transitive Hülle (vgl. [27]) einer zweistelligen Relation R, notiert mit transhull(R), totale Ordnung (vgl. [26])
- Definition, Variable in einer Definition (vgl. [28])
- Prädikatenlogik (vgl. [29]) – wobei Implikationen mit $\Longrightarrow$, Äquivalenzen mit $\Longleftrightarrow$ und die Menge der Wahrheitswerte mit BOOL notiert wird, wobei BOOL = { true, false }.
- Induktion, induktive Definition (vgl. [30])
- Spezifikation (vgl. [31])

Folgende Rangfolge (beginnend mit dem höchsten Rang) von Operationen sei vereinbart:

- Kartesisches Produkt: $\times$
- Vereinigung, Schnitt und Subtraktion: $\cup, \cap, /$
- Teilmengen- und Element-Beziehungen: $\subseteq, \in, \notin$
- Gleichheit: =
- Konjunktion: $\wedge$
- Implikation und Äquivalenz: $\Longrightarrow, \Longleftrightarrow$
- Zuordnung: $\mapsto$

Für alle weiteren Operationen wird die Rangfolge durch Klammerung explizit angegeben.

Weiterhin seien folgende Notationen vereinbart[2], wobei das Symbol $\Leftrightarrow$ für „entspricht“ steht:

- $\bigcup x \in \{x1, \ldots, xn\} : f(x) \quad \Leftrightarrow \{f(x1)\} \cup \ldots \cup \{f(xn)\}$
- $\bigcup x \in M \text{ mit } p(x) : f(x) \quad \Leftrightarrow \bigcup x \in \{ x \in M \mid p(x) \} : f(x)$
- $\sum x \in \{x1, \ldots, xn\} : f(x) \quad \Leftrightarrow f(x1) + \ldots + f(xn)$
- $\forall x \in M \text{ mit } p(x), q(x): A(x) \quad \Leftrightarrow \forall x \in M : p(x) \wedge q(x) \Longrightarrow A(x)$
- $\forall x \in M, y \in N : A(x,y) \quad \Leftrightarrow \forall x \in M : [\forall y \in N : A(x,y)]$

Definition 2.1: Beliebiges Element aus einer Menge
Ist eine Menge M *gegeben, so ist die Funktion* any *wie folgt definiert:*

$$\text{any}(M) =_{\text{def}} \text{falls } M=\emptyset \text{ dann n.def., sonst beliebiges Element aus } M$$

[1] Das hier verwendete Begriffsverständnis orientiert sich an [24], nach dem es „[bei] der Operationalisierung [um] genaue („technische“) Anweisungen [geht,] wie im konkreten Fall vorzugehen ist, um die gewünschte Information aus der Realität zu erhalten.“. Obwohl dieser Begriff in [24] im Bezug zur Empirie steht, wird in dieser Arbeit der allgemeinere Rahmen verwendet, beliebige Information der Realität greifbar zu machen.

[2] Für die Definition einiger Notationen erfolgt ein Vorgriff auf das Konzept einer Funktion aus Kapitel 2.1.1.

2.1.1 Funktion

Definition 2.2: Funktion, Definitionsbereich, Wertebereich
*Eine **Funktion** ist eine, über die Zeit hinweg ggf. nicht konstante Relation zwischen zwei Mengen* D *und* W*, wobei jedes Element aus* D *maximal zu einem* W *in Beziehung steht. Zur Übersichtlichkeit sei bezüglich des Zeitbegriffes der Vergleich zu Funktionen in Programmiersprachen gezogen. Ist eine Funktion* f *gegeben, so sind dessen* **Definitionsbereich***, notiert als* DB(f)*, und* **Wertebereich***, notiert als* WB(f)*, wie folgt definiert:*

$$DB(f) =_{def} \bigcup_{(x,y)\in f} \{x\}$$
$$WB(f) =_{def} \bigcup_{(x,y)\in f} \{y\}$$

Ist ein $a \in DB(f)$ *gegeben, so ermittelt sich das zu* a *durch* f *zugeordnete Element aus* WB(f)*, notiert als* f(a)*, wie folgt*:

$$f(a) =_{def} \text{falls } |\{(x,y)\in WB(f) | x=a\}|=1 \text{ dann any}(\{(x,y)\in WB(f)|x=a\}) \text{ sonst n.def.}$$

Für eine kompakte Notation von Funktionen sei folgendes vereinbart:

$$\{ (k1,w1), \ldots, (kn,wn) \} \Leftrightarrow \{ k1 \mapsto w1, \ldots, kn \mapsto wn \}$$

Ist eine Funktion f *und die Mengen* D *und* W *gegeben, und gilt* $DB(f) \subseteq D$ *und* $WB(f) \subseteq W$*, so wird dies notiert als:*

$$f : D \rightarrow W$$

Die Menge aller Funktionen f *mit* $f : D \rightarrow W$ *wird wie folgt notiert:*

$$\{ f : D \rightarrow W \}$$

Definition 2.3: Definiertheit
Ist eine Funktion f *und ein mathematisches Objekt* $e \in DB(f)$ *gegeben, so wird* f(e) *als **definiert** bezeichnet. Falls jedoch* $e \notin DB(f)$ *gilt, wird* f(e) *als **nicht definiert** bezeichnet. Ist* f(e) *nicht definiert, und h eine beliebige Funktion, dann ist auch* h(f(e)) *nicht definiert.*

Definition 2.4: Beschränkung einer Funktion
*Die **Beschränkung** einer Funktion* f *auf eine Menge* X*, notiert als* $f \restriction X$*, ist wie folgt definiert:*

$$f \restriction X =_{def} \{ (x,y)\in f \mid x \in X \}$$

Definition 2.5: Zusammenhang Funktion und Prädikat
Eine Funktion p *wird als Prädikat bezeichnet, wenn sie auf Wahrheitswerte abbildet, dh.* $WB(p) \subset BOOL$*. Zur Hervorhebung von Prädikaten werden bei der Anwendung als Notation spitze statt runde Klammern verwendet:*

$$p<a1\ldots an> \Leftrightarrow p(a1\ldots an)$$

Ein Prädikat $p : D \rightarrow BOOL$ kann auch als eine Teilmenge d von D verstanden werden, mit $d = \{ x\in D \mid p<x>=true \}$. Umgekehrt kann zu einer Teilmenge d stets eine Funktion p durch $p(x) =_{def} x \in d$ konstruiert werden, die auf Wahrheitswerte abbildet.

Definition 2.6: Referenzielle Transparenz
Sei eine Funktion f *und ein* x ∈ DB(f) *gegeben. Wenn* f(a) *unabhängig vom Auswertungszeitpunkt ist, wird* f *als* ***referenziell transparent*** *bezeichnet, andernfalls als* ***referenziell nicht transparent***.

Anders formuliert liefert eine referenziell transparente Funktion bei jeder Auswertung für dieselben Parameter dasselbe Ergebnis. Bei einer referenziell nicht transparenten Funktion kann das Ergebnis hingegen von Faktoren abhängen, die nicht als Parameter erfasst sind. Dadurch können unterschiedliche Auswertungen mit denselben Parametern zu unterschiedlichen Ergebnissen führen. Ein Beispiel einer referenziell nicht transparanten Funktion ist eine Eingabeaufforderung an einen Anwender: Aus Sicht des Systems kann das vom Benutzer gelieferte Ergebnis nicht allein anhand des dem Benutzer gegebenen Parameter (wie z.B. die Ausgabe auf der Graphischen Benutzungsschrnittstelle) beschrieben werden.

2.1.2 Struktur

Zur algebraischen Definition von Mengen mit zusammengesetzten Elementen werden üblicherweise Tupel verwendet. Beispielsweise wird in der Mathematik ein Graph als ein 2-Tupel (N,E) definiert, wobei N gleich der Menge der Knoten des Graphen ist und E eine Teilmenge von N×N, die Menge der Kanten des Graphen ist. Dabei sind also alle Bestandteile eines Graphen allgemein bekannt, und der Zugriff auf einen Bestandteil erfolgt über die Projektion unter Angabe der Position (als eine natürliche Zahl) – z.B. $\pi_1(g)$ für die Knoten, und $\pi_2(g)$ für die Kanten des Graphen g.

Der hier definierte Begriff einer Struktur ermöglicht es, Definitionen so zu formulieren, dass die Bestandteile der definierten Elemente nicht im vollen Umfang bekannt sein müssen. Damit kann Grundlegendes definiert werden (wie Graphmorphismen, um am Beispiel von Graphen zu bleiben), ohne dass die Definition formal gesehen nicht mehr zutreffen, sobald erweiternde Konzepte (wie getypte Graphen) eingeführt werden (für die ein Graph neben Knoten und Kanten noch weitere Bestandteile enthalten muss). Durch Verwendung von Bestandteilssymbolen wird es zudem möglich, erweiternde Konzepte auch formal ohne Weiteres beliebig zu kombinieren, da zusätzliche Bestandteile nicht – wie bei Tupeln bezüglich der Position im Tupel – kollidieren können.

Definition 2.7: Bestandteilssymbol
Die Menge PSYMBOL *ist die Menge aller* ***Bestandteilssymbole****. Im Rahmen dieser Arbeit ist diese Menge wie folgt festgelegt:*

PSYMBOL $=_{def}$ { Id, s, t, y, add, del, dom, ign, term, Cnst, neg, idm, inherit, idmap, hist}

Die Bedeutung der einzelnen Bestandteilssymbole wird an gegebener Stelle erläutert.

Definition 2.8: Struktur
Eine Funktion g *mit* DB(g) ⊂ PSYMBOL *wird als* ***Struktur*** *bezeichnet. Die Menge aller Strukturen wird mit* STRUCT *notiert. Sind eine Struktur* g ∈ STRUCT *und ein* b ∈ PSYMBOL *gegeben, und ist* g(b) *definiert, wird dies mit „*g ***hat*** *den* ***Bestandteil*** b*“ ausgedrückt. Der konkrete Wert von* g(b) *wird als „****der Bestandteil*** b ***von*** g*“ bezeichnet, und kurz mit* b_g *notiert. Sollte* g(b) *nicht definiert sein, so liefert* b_g *die leere Menge:*

$b_g =_{def}$ falls g(b) def. dann b(g), sonst ∅

Definition 2.9: Substitution
Gegeben sei eine Struktur $g \in STRUCT$*, ein Bestandteilssymbol* $b \in PSYMBOL$ *und ein beliebiges mathematisches Objekt* w*. Die* ***Substitution*** *des Bestandteils* b *von* g *durch* w*, notiert als* $g[b \mapsto w]$*, ist definiert durch:*

$$g[b \mapsto w] =_{def} (g / \{b \mapsto b_g\}) \cup \{b \mapsto w\}$$

2.1.3 Formale Sprache

Definition 2.10: Wort, Wortraum
Ein ***Wort*** *ist ein Element einer beliebig gewählten Menge die als die Menge aller Wörter angesehen wird. Im Folgenden wird die gewählte Menge aller Wörter als* ***Wortraum*** *bezeichnet.*

Beispielsweise kann der Wortraum der Menge aller Graphen (bei graphbasierten Sprachen) oder der Menge aller Zeichenketten (bei textbasierten Sprachen) gleichgesetzt werden.

Definition 2.11: Wortsyntax
Eine ***Wortsyntax*** *ist eine endliche Darstellung eines gegebenenfalls unendlich großen Wortraums.*

Bei textbasierten Sprachen wird der Wortraum beispielsweise durch ein endliches Alphabet von Zeichen dargestellt: Der Wortraum entspricht dann der Menge aller Zeichenketten, die sich durch beliebig häufige Aneinanderreihung von Zeichen aus dem Alphabet ergeben. Bei graphbasierten Sprachen ist dies analog möglich – mit dem Unterschied, dass aus den Zeichen keine Ketten, sondern Graphen gebildet werden können.

Definition 2.12: Formale Sprache
Eine ***formale Sprache*** *ist eine Teilmenge eines Wortraums.*

Wird als Wortraum die Menge aller Zeichenketten verwendet, ist Java [32] eine formale Sprache: Jedes Java-Programm ist eine Zeichenkette – und damit Element des Wortraums. Da umgekehrt nicht jede Zeichenkette ein Java-Programm ist, bildet Java eine echte Teilmenge des Wortraums (was jedoch keine notwendige Bedingung für eine formale Sprache ist).

Definition 2.13: Syntax, Extensionsfunktion, Extension
Eine ***Syntax*** *ist eine endliche Darstellung einer (ggf. unendlich großen) formalen Sprache. Die* ***Extensionsfunktion*** *ist eine Funktion die zu einer Syntax als Eingabe, die durch sie beschriebene formalen Sprache als Ausgabe liefert. Die Ausgabe wird auch als Extension bezeichnet. Die* ***Extension*** *einer Syntax* S *wird als* $Ext(S)$ *notiert.*

Bei unendlichen Sprachen ist die Extensionsfunktion nicht terminierend ausführbar. Dennoch kann sie, unter bestimmten Voraussetzungen (siehe Wortproblem, vgl. [33]) genutzt werden, um für ein konkretes Wort zu bestimmen, ob es zu der von ihr beschriebenen Sprache gehört. Beispielsweise kann anhand der Sprachdefinition von Java ermittelt werden, ob eine vorliegende Zeichenkette tatsächlich ein gültiges Java-Programm ist, ohne zunächst die Extension vollständig ermitteln zu müssen.

Definition 2.14: Semantik
*Eine **Semantik** ist eine beliebige Funktion mit einer Sprache als Definitions- und einer beliebigen Menge als Wertebereich.*

Durch die Semantik einer formalen Sprache wird jedem Wort eine Bedeutung zugewiesen. In der Regel hat eine formale Sprache genau eine Semantik. Beispielsweise ist der Programmiersprache Java durch die JavaRuntimeEnvironment eine operationelle Semantik zugeordnet, die jedem Java-Programm (Wort) durch Ausführen einen Systemzustand zuweist. Die Semantik ist die Grundlage für denjenigen, der ein Wort zu dieser Sprache erstellt. In Fortsetzung des Java-Beispiels muss ein Programmierer wissen, wie Sprachkonstrukte von Java verarbeitet werden, um ein Programm schreiben zu können das das von ihm beabsichtigte Verhalten hat.

Definition 2.15: Abstrakte und Konkrete Syntax
Sind eine formale Sprache L*, also eine ggf. unendlich große Menge von Wörtern, sowie zwei Syntaxen* KS *und* AS *mit* Ext(KS)=L, |Ext(AS)|=|L| *gegeben, wird* KS *als eine **konkrete Syntax** von* L *und* AS *als eine **abstrakte Syntax** von* L *bezeichnet.*

Als Beispiel seien natürliche Zahlen betrachtet, für die es verschiedene Sprachen gibt: „arabisch“ (z.B. 321), „römisch“ (CCCXXI), „binär“ (1010000011) oder „ausgeschrieben“ (Dreihunderteinundzwanzig). Die Sprachen haben je Anwendungsfall unterschiedliche Vor- und Nachteile. So ist die arabische Darstellung aus dem praktischen Leben nicht mehr wegzudenken, während die binäre Darstellung die automatisierte Verarbeitung und sicheren Transport der Daten vereinfacht. Die konkrete Syntax beschreibt eine Sprache, die vom Anwender tatsächlich zur Eingabe oder Bearbeitung von Wörtern verwendet wird. Sie ist daher im Hinblick auf Benutzungsaspekte wie Lesbarkeit, Intuitivität und/oder schnelle Eingabe ausgewählt bzw. optimiert. Die abstrakte Syntax beschreibt eine Sprache, die auf die Verarbeitung von Wörtern optimiert ist. Sie ist daher z.B. auf die Vereinfachung zur Definition einer Semantik ausgerichtet. Ein mit einer konkreten Syntax eingegebenes Wort verarbeiten zu können, muss dieses durch eine Funktion in ein Wort der abstrakten Syntax überführt werden. Sofern das nach Anwendung der Semantik entstehende Wort dem Benutzer anzuzeigen ist, ist eine entgegengerichtete Funktion notwendig. Der Einfachheit halber wird in dieser Definition davon ausgegangen, dass beide Funktionen existieren müssen, und die Extensionen beider Darstellungen (konkret und abstrakt) umfassen. Daraus ergibt sich die Forderung, dass beide Extensionen gleich groß zu sein haben.

2.2 Graphen und Transformationen

Graphbasierte formale Sprachen sind formale Sprachen bei denen Wörter Graphen sind, die wiederum aus beschrifteten Ecken und beschrifteten Kanten zusammengesetzt sind.

2.2.1 Graphen als Struktur für Wörter

Bevor die Struktur des Graphen definiert wird, seien zunächst die Mengen aller Knoten und Kanten, die in ihrer Vereinigung[3] die Menge GE aller Graphelemente bilden, definiert.

Definition 2.16: Graphelement, Id
Die Menge GE *aller* ***Graphelemente*** *ist wie folgt definiert:*

$$GE =_{def} \text{Menge der Bezeichner für Knoten und Kanten}$$

Ein Graphelement wird auch als eine ***Id*** *bezeichnet.*

Definition 2.17: Graph
Die Menge GRAPH *aller* ***Graphen*** *ist wie folgt definiert:*

$$\begin{aligned} GRAPH =_{def} \{\, g \in STRUCT \mid\ & \underline{Id}_g \subseteq GE \wedge \\ & \underline{s}_g \in \{ f : GE \rightarrow GE \} \wedge \\ & \underline{t}_g \in \{ f : GE \rightarrow GE \} \wedge \\ & [\forall i \in \underline{Id}_g : \underline{s}_g(i) \text{ def.} \iff \underline{t}_g(i) \text{ def.}] \wedge \\ & [\forall i \in \underline{Id}_g : \underline{s}_g(i) \text{ def.} \wedge \underline{t}_g(i) \text{ def.} \implies i \notin WB(\underline{s}_g) \cup WB(\underline{t}_g)\ ] \,\} \end{aligned}$$

Zur einfacheren Handhabung seien folgende Hilfsfunktionen vereinbart:

$$\begin{aligned} nodes(\, g \,) &=_{def} \{\, i \in \underline{Id}_g \mid \underline{s}_g(i) \text{ n.def.} \wedge \underline{t}_g(i) \text{ n.def.} \,\} \\ edges(\, g \,) &=_{def} \{\, i \in \underline{Id}_g \mid \underline{s}_g(i) \text{ def.} \wedge \underline{t}_g(i) \text{ def.} \,\} \end{aligned}$$

Die ersten drei Bedingungen formulieren die Bestandteile, die ein Graph g mindestens hat:

- $\underline{Id}_g \subset GE$ – Eine Menge der Graphelemente des Graphen
- $\underline{s}_g$ – Eine Funktion, die einem als Kante angesehenen Graphelement genau einen als Startknoten angesehenes Graphelement zuweist
- $\underline{t}_g$ – Eine Funktion die einem als Kante angesehenen Graphelement genau einen als Zielknoten angesehenes Graphelement zuweist

Die vierte Bedingung besagt, dass für ein Graphelement stets entweder Start- und Zielknoten definiert sind (wodurch das Graphelement als eine Kante angesehen wird) oder keines davon (wodurch es als ein Knoten angesehen wird). Die letzte Bedingung fordert, dass eine Kante nicht von einer anderen referenziert werden kann.

Die Funktion nodes liefert die als Knoten angesehenen Graphelemente eines Graphen, und edges die als Kanten angesehenen Graphelemente.

Definition 2.18: Wohlgeformtheit eines Graphen
Ein Graph $g \in$ GRAPH *wird als* ***wohlgeformt*** *bezeichnet, notiert als* wellformed<g>*, genau dann, wenn alle referenzierten Graphelemente auch Teil des Graphen sind*:

$$wellformed\langle g \rangle =_{def} (WB(\underline{s}_g) \subseteq nodes(g)) \wedge (WB(\underline{t}_g) \subseteq nodes(g)) \wedge (DB(\underline{s}_g) \subseteq \underline{Id}_g) \wedge (DB(\underline{t}_g) \subseteq \underline{Id}_g)$$

[3] Knoten und Kanten sind hier also nicht anhand ihrer Id unterscheidbar

Definition 2.19: Addition und Subtraktion von Graphen
Gegeben seien zwei Graphen g1, g2 ∈ GRAPH. *Die* ***Addition*** *von* g1 *und* g2*, notiert als* g1 + g2*, und die* ***Subtraktion*** *von* g1 *und* g2*, notiert als* g1 – g2*, sind wie folgt definiert:*

$$g1 + g2 =_{def} \bigcup b \in DB(g1): \{ b \mapsto b_{g1} \cup b_{g2} \}$$
$$g1 - g2 =_{def} \bigcup b \in DB(g1): \{ b \mapsto b_{g1} / b_{g2} \}$$

Zu jedem Bestandteilssymbol b von g1 wird zu dem Bestandteil b_{g1} der entsprechende Bestandteil von g2, b_{g2}, durch eine Vereinigung hinzugefügt. Diese Fassung der Addition ist nicht kommutativ, da Bestandteile, die in b2 aber nicht in b1 vorkommen, verworfen werden. Weiterhin ist das Ergebnis nur dann ein Graph, wenn für g1 und g2 gilt:

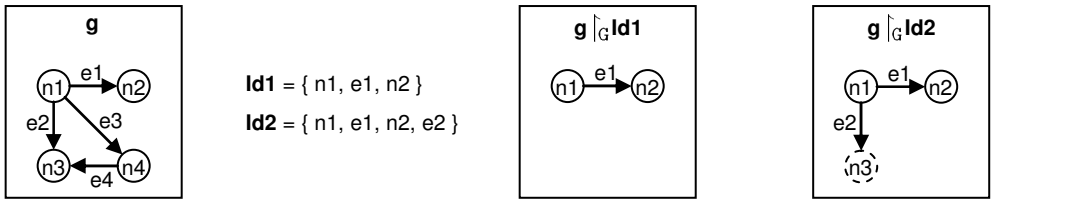
$$edges(g1) \cap nodes(g2) = \emptyset \wedge$$
$$nodes(g1) \cap edges(g2) = \emptyset \wedge$$
$$((WB(\underline{t}_{g1}) \cup WB(\underline{s}_{g1})) \cap edges(g2) = \emptyset \wedge$$
$$(edges(g1) \cap (WB(\underline{t}_{g2}) \cup WB(\underline{s}_{g2})) = \emptyset$$

Ids die in einem Graph für Knoten stehen, dürfen im anderen Graph nicht für Kanten stehen. Die Ids der Start- und Zielknoten von Kanten des einen Graphen dürfen nicht für Knoten im anderen Graphen stehen. Diese Bedingung wird nicht durch die vorherige impliziert, da hier die Addition nicht-wohlgeformter Graphen zugelassen ist.

Definition 2.20: Beschränkung eines Graphen auf Graphelemente
Gegeben sei ein Graph g ∈ GRAPH *und eine Menge von Graphelementen* Id ⊆ GE. *Die* ***Beschränkung*** *des Graphen* g *bezüglich* Id*, notiert als* $g \restriction_G Id$*, ist wie folgt definiert*:

$$g \restriction_G Id =_{def} g[\ \underline{Id} \mapsto \underline{Id}_g \cap Id\ ][\ \underline{s} \mapsto \underline{s}_g \restriction Id\ ][\ \underline{t} \mapsto \underline{t}_g \restriction Id\ ]$$

In $g \restriction_G Id$ verbleiben nur Graphelemente, die auch in Id vorkommen. Für so verschwindende Kanten werden auch die Zuordnungen der Start- und Zielknoten entfernt. Abbildung 2.1 zeigt dazu zwei Beispiele: Der Graph g wird durch Id1 auf die Graphelemente n1, e1 und n2 beschränkt. Die Beschränkung auf Id2 liefert einen ähnlichen, aber nicht wohlgeformten Graphen, da der von e2 referenzierte Knoten n3 nicht im Bestandteil Id des resultierenden Graphen vorkommt. In Darstellungen werden solche Knoten mit unterbrochener Linie notiert.



$\underline{Id}_g$ = { n1, n2, n3, n4 }
$\underline{s}_g$ = { e1↦n1, e2↦n1, e3↦n1, e4↦n4 }
$\underline{t}_g$ = { e1↦n2, e2↦n3, e3↦n4, e4↦n3 }

$\underline{Id}_{g|_G Id1}$ = { n1, n2 }
$\underline{s}_{g|_G Id1}$ = { e1↦n1 }
$\underline{t}_{g|_G Id1}$ = { e1↦n2 }

$\underline{Id}_{g|_G Id2}$ = { n1, n2 }
$\underline{s}_{g|_G Id2}$ = { e1↦n1, e2↦n1 }
$\underline{t}_{g|_G Id2}$ = { e1↦n2, e2↦n3 }

Abbildung 2.1 Beschränkung eines Graphen

2.2.2 Graphhomomorphismus

Ein Graphhomomorphismus ist eine Funktion, die einen gegebenen Graphen g1 strukturerhaltend auf einen (Teil eines) gegebenen Graphen g2 abbildet (vgl. Abbildung 2.2). Anders formuliert wirkt eine solche Funktion als eine „Umbenennung“ von Knoten und Kanten. Sie kann also weder Knoten hinzufügen ooder entfernen, noch die Zuordnung der Start- und Zielknoten vorhandener Kanten ändern.

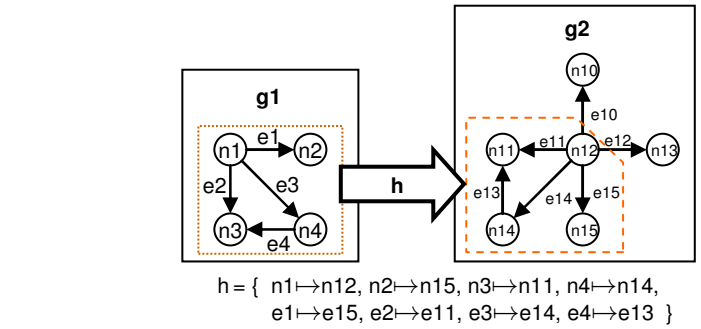


Abbildung 2.2 Graphhomomorphismus

Definition 2.21: Graphmorphismus
Die Menge GRAPHMORPH *aller* ***Graphmorphismen*** *ist wie folgt definiert:*

$$\text{GRAPHMORPH} =_{def} \{ h \mid h : GE \rightarrow GE \}$$

Um einen Graphmorphismus h *als Funktion auf einen Graphen* g ∈ GRAPH *formal korrekt anwenden zu können, sei folgendes vereinbart:*

$$h(g) =_{def} g' = g[Id \mapsto Id'][s \mapsto s'][t \mapsto t'] \text{ mit}$$
$$Id' = \bigcup_{i \in Id} h'(i)$$
$$s' = \bigcup_{(e,n) \in s_g} \{ (h'(e), h'(n)) \}$$
$$t' = \bigcup_{(e,n) \in t_g} \{ (h'(e), h'(n)) \}$$
wobei
$$h' = h \cup \{ (i,i) \in Id_g \times Id_g \mid i \notin DB(h) \}$$

Graphmorphismen sind als Funktionen über Graphelementen definiert. Das Anwenden eines Graphmorphismus h auf einen Graph g erzeugt einen neuen Graph g’. Dazu wird jedes Graphelement, jede Start- und jede Zielzuordnung durch h einzeln abgebildet. Um im Allgemeinen auch den Fall abzudecken, dass durch h nicht alle Graphelemente von g abgedeckt werden, wird die Hilfsfunktion h’ genutzt, die alle Graphelemente auf sich selbst abbildet, die nicht durch h abgebildet werden. Durch Nutzung der Substitution werden alle anderen Bestandteile von g unverändert nach g’ übernommen. Dieses Verhalten wird jedoch für erweiterte Graphstrukturen (z.B. getypte Graphen) überschreiben.

Definition 2.22: Graphhomomorphismus
Seien die Graphen g1, g2 ∈ GRAPH *und ein Graphmorphismus* h ∈ GRAPHMORPH *gegeben. Falls* h *alle Graphelemente aus* g1 *strukturerhaltend auf die aus* g2 *abbildet, wird* h *als* ***Graphhomomorphismus*** *bezüglich* g1 *und* g2 *bezeichnet, diese Situation als* homomorph<h,g1,g2> *notiert, und ist wie folgt definiert*:

homomorph<h,g1,g2> =$_{def}$
DB(h) = $\underline{ld}_{g1}$ ∧
[∀ e∈edges(g1): h(e) ∈ edges(g2)] ∧
[∀ n∈nodes(g1): h(n) ∈ nodes(g2)] ∧
[∀ e∈edges(g1): h($\underline{s}_{g1}$(e)) = $\underline{s}_{g2}$(h(e))] ∧
[∀ e∈edges(g1): h($\underline{t}_{g1}$(e)) = $\underline{t}_{g2}$(h(e))]

Durch die erste Bedingung wird gefordert, dass der Graphmorphismus h für jedes Graphelement von g1 definiert ist, d.h., dass der gesamte Graph g1 abgebildet wird. Die folgenden beiden Bedingungen stellen sicher, dass ein Graphmorphismus Knoten immer auf Knoten, und Kanten immer auf Kanten abbildet. Durch die letzten beiden Bedingungen wird sichergestellt, dass Startknoten und Zielknoten zur Struktur passend abgebildet werden.

2.2.3 Produktionsregel

Eine Produktionsregel ist ein Suche-nach-X-und-ersetze-durch-Y-Muster welches verwendet werden kann um Graphen zu verändern. Wird eine solche Regel auf einen gegebenen Graphen g angewendet, werden in g alle Redexe (Teilgraphen aus g die dem Suchmuster X gleichen) durch Y ersetzt. Graphelemente die sowohl im Suchmuster X als auch im Ersetzungsmuster Y vorkommenwerden effektiv nicht verändert.

Bei der Kompaktdarstellung werden das Such- und das Ersetzungsmuster nicht als zwei eigenständige Graphen geführt, sondern in einem gemeinsamen Graphen zusammengefasst. Um zu unterscheiden, welche Teile des Graphen für die Suche (bzw. das Ersetzen) relevant sind, werden Markierungen an die Knoten und Kanten angebracht. Elemente die zum Kontext gehören, also die beide Markierungen haben, werden zur Übersichtlichkeit ganz ohne Markierungen dargestellt (die sich effektiv auch aufheben). Abbildung 2.3 zeigt eine Produktionsregel in drei Darstellungen: kompakt, als zwei eigenständige Such- und Ersetzungsmuster sowie als eine solche, die das Muster für den Kontext aus dem Such- und Ersetzungsmuster herausnimmt und als dritten eigenständigen Graph führt. Das um den Kontext erleichterte Suchmuster wird als Löschmuster und das um den Kontext erleichterte Ersetzungsmuster als Erzeugungsmuster bezeichnet.

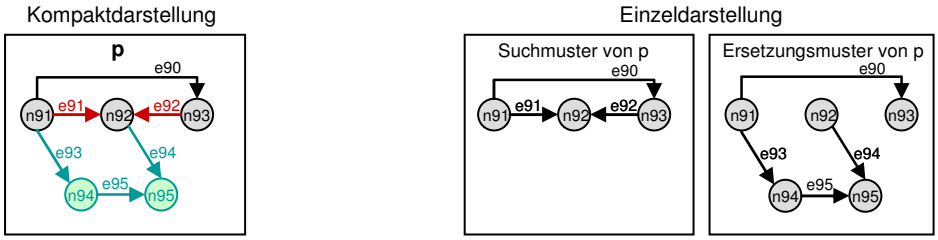


Abbildung 2.3 Kompakt- und Einzeldarstellung einer Produktionsregel

Zu beachten ist, dass die über einer Produktionsregel definierten Teilmuster im Allgemeinen keine wohlgeformten Graphen ergeben, da die Werte- und Definitionsbereiche der $\underline{s}$- und $\underline{t}$-Bestandteile im Allgemeinen größer sind als die Knoten- und Kantenmengen.

Definition 2.23: Produktionsregel, Musterelemente, Suchmuster, Ersetzungsmuster
Die Menge PRODRULE *aller* ***Produktionsregeln*** *ist wie folgt definiert:*

PRODRULE =$_{def}$ { p ∈ GRAPH |
$\underline{add}_p \subseteq \underline{ld}_p$ ∧
$\underline{del}_g \subseteq \underline{ld}_p$ ∧
$\underline{add}_p \cap \underline{del}_p = \emptyset$ ∧
wellformed< p > ∧
wellformed< p $|_G$ ($\underline{ld}_p$/$\underline{add}_p$) > ∧

wellformed< p $\restriction_G$ (ld_p/del_p) > }

Die Graphelemente einer Produktionsregel werden als ***Musterelemente****, und entsprechend Knoten als* ***Musterknoten*** *sowie Kanten als* ***Musterkanten*** *bezeichnet. Für ein* p ∈ PRODRULE *sind das* ***Suchmuster*** *von* p*, notiert mit* searchpattern(p)*, und das* ***Ersetzungsmuster****, notiert mit* replacepattern(p)*, wie folgt definiert:*

searchpattern(p) $=_{def}$ p $\restriction_G$ (ld_p/add_p)
replacepattern(p) $=_{def}$ p $\restriction_G$ (ld_p/del_p)

Eine Produktionsregel ist eine Struktur, die als Graph mindestens die Bestandteile ld, s und t hat. Darüber hinaus enthält eine Produktionsregel p die Bestandteile add und del, wobei

- $add_p \subseteq ld_p$ – ein Prädikat über den Musterelementen der Produktionsregel welches angibt, welche Graphelemente bei Anwendung der Produktionsregel auf einen Graphen zum selbigen hinzukommen. In Abbildung 2.3 sind die Musterelemente grün gekennzeichnet.
- $del_p \subset ld_p$ – ein Prädikat über den Musterelementen der Produktionsregel welches angibt, welche Graphelemente bei Anwendung der Produktionsregel auf einen Graphen aus selbigen gelöscht werden. In Abbildung 2.3 sind diese rot gekennzeichnet.

Graphelemente können nicht gleichzeitig als hinzuzufügen und als zu löschen markiert werden. Zudem muss die Produktionsregel sowohl als ganzes, als auch beschränkt auf dessen Such- oder Ersetzungsmuster, wohlgeformt sein.

Definition 2.24: Redexmorphismus, Ersetzungsmorphismus
Die Menge REDEXMORPH *aller* ***Redexmorphismen*** *und die Menge* REPLACEMORPH *aller* ***Ersetzungsmorphismen*** *sind wie folgt definiert:*

REDEXMORPH $=_{def} \bigcup$ p∈PRODRULE, g∈GRAPH:
{ hx∈GRAPHMORPH | homomorph<hx,searchpattern(p),g> }

REPLACEMORPH $=_{def} \bigcup$ p∈PRODRULE, g∈GRAPH:
{ hx∈GRAPHMORPH | homomorph<hx,replacepattern(p),g> }

Ein Redexmorphismus hx beschreibt, wie das Suchmuster einer Produktionsregel p auf einen dazu passenden Teil – den Redex – in einem Graphen g abgebildet werden kann. Der Ersetzungsmorphismus beschreibt, wie das Ersetzungsmuster von p zur Ersetzung des Redex einzusetzen ist.

Definition 2.25: Patternmatchingfunktion
Die ***Patternmatchingfunktion*** pmf *ist eine Funktion, die zu einem Graph* g ∈ GRAPH *und einer Produktionsregel* p ∈ PRODRULE *als Eingabe, die Menge aller Redexmorphismen bezüglich* g *und* p *als Ausgabe liefert, und wie folgt spezifiziert ist:*

∀ g∈GRAPH, p∈PRODRULE:
pmf(g,p) $=_{spec}$ { hx ∈ REDEXMORPH | homomorph<hx,searchpattern(p),g> }

Eine konkrete Berechnungsvorschrift für diese Funktion ist für das Weitere nicht relevant.

Definition 2.26: Id-Generierungsfunktion
*Die **Id-Generierungsfunktion*** newGE *ist eine nicht referenziell transparente Funktion die bei jeder Ausführung ein neues, bisher nicht verwendetes Element aus* GE *zurückliefert, und wie folgt spezifiziert ist:*

$$\text{newGE} : \rightarrow \text{GE}$$
$$\text{newGE}() \neq_{spec} \text{newGE}()$$

Die Funktion newGE hat keinen Parameter. Die Spezifikationsbedingung sagt aus, dass zwei Aufrufe dieser Funktion nicht das gleiche Ergebnis liefern.

Definition 2.27: Erzeugungsfunktion
*Die **Erzeugungsfunktion*** rmf *ist eine referenziell nicht transparente Funktion die zu einem Graph* g ∈ GRAPH*, einer Produktionsregel* p ∈ PRODRULE *und einem Redexmorphismus* h ∈ REDEXMORPH *bezüglich* p *und* g *als Eingabe, einen Ersetzungsmosphismus aus* REPLACEMORPH *als Ausgabe liefert, und wie folgt spezifiziert ist:*

$$\forall\, p \in \text{PRODRULE}, g \in \text{GRAPH}, hx \in \text{REDEXMORPH}:$$
$$\text{rmf}(p,g,hx) =_{spec} \text{any}(\{\ hc \in \text{REPLACEMORPH} \mid$$
$$\text{homomorph}\langle hc,\text{replacepattern}(p)\rangle,g\rangle \wedge$$
$$[\forall\, i \in \text{DB}(hx) \cap \text{DB}(hc) : hx(i)=hc(i)] \wedge$$
$$[\ \bigcup\, i \in \text{DB}(hc)/\text{DB}(hx) : \{\ hc(i)\ \}\ ] \cap V = \emptyset$$
$$\})$$

wobei
V die Menge aller vor der Ausführung von rmf verwendeten Graphelemente umfasst

Die erste Bedingung in der Spezifikation fordert, dass hc ein zu p und g passender Ersetzungsmorphismus ist. Dabei wird durch die zweite Bedingung zusätzlich sichergestellt , dass die Belegungen derjenigen Musterlemente gleich sind, die sowohl im Such- als auch Ersetzungsmuster vorkommen. Die dritte Bedingung fordert, dass alle durch die Ersetzung zu g hinzukommenden Graphelemente völlig neu sein müssen. Eine Beschränkung auf das Nichtenthaltensein in g würde hier zu kurz kommen, da eine hier ausgeführte Erzeugung Bestandteil einer umgebenden, größeren sein kann. Auch die dort bereits existierenden Graphelemente dürfen nicht als neu erzeugte verwendet werden. Aus dieser Anforderung folgt, dass die Funktion nicht referenziell transparent (selbst bei der gleichen Eingabe g sind die Ergebnisse zweier nacheinander folgender Ausführungen rmf(g) nicht identisch). Eine konkrete Berechnungsvorschrift für diese Funktion ist für das Weitere nicht relevant. Zur Umsetzung eignet sich jedoch die Nutzung der Funktion newGE.

Definition 2.28: Reduktion
Gegeben seien ein Graph g ∈ GRAPH*, eine Produktionsregel* p ∈ PRODRULE *und ein Redexmorphisms* hx ∈ REDEXMORPH. *Die **Reduktion** eines Graphen* g *bezüglich* p *und* hx*, notiert als* reduce(g,p,hx)*, liefert einen neuen Graph* g' ∈ GRAPH *und den dazu verwendeten Ersetzungsmorphismus* hc ∈ REPLACEMORPH *ist wie folgt definiert:*

$$\text{reduce}(g,p,hx) =_{def} (g'',hc) = (\ [\ g - (\text{redex}/\text{replacement})\ ] + (\text{replacement}/\text{redex})\ ,\ hc\ ) \text{ mit}$$
$$\text{redex} = hx(\text{searchpattern}(p))$$
$$\text{replacement} = hc(\text{replacepattern}(p))$$

wobei
hc = rmf(p,g,hx), einmalig ausgeführt

Durch die Reduktion wird aus dem Graph g ein neuer Graph g', wie in Abbildung 2.4 an einem Beispiel veranschaulicht. Vom Ablauf her wird zunächst der Ersetzungsmorphismus hc

durch einmalige Ausführung von rmf berechnet. Aus der Differenz des Such- und Ersetzungsmusters (hx(searchpattern(p)/hc(replacepattern)) ergeben sich die aus g zu löschenden Graphelemente. Aus der dazu dualen Differenz (hc(replacepattern)/hx(searchpattern(p)) ergeben sich die zu g hinzuzufügenden Graphelemente.

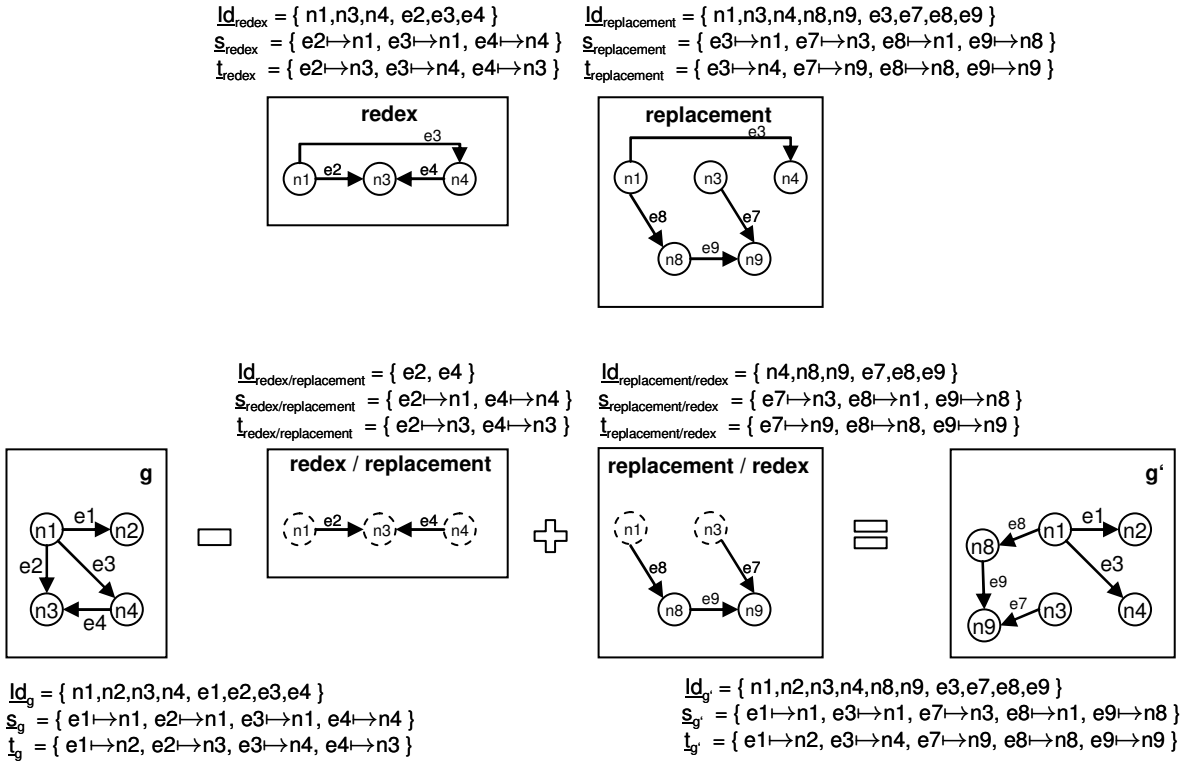


Abbildung 2.4 Reduktion eines Graphen

Zur einfacheren Formulierung auf der Reduktion aufbauender Definitionen, wird der verwendete Ersetzungsmorphismus hc als ein weiterer Parameter zurückgegeben.

2.2.4 Graphgrammatik und Transformation

Bisher wurde nur eine einmalige Ausführung einer einzelnen Produktionsregel auf einen Graphen betrachtet, und dies als Reduktion bezeichnet. Wird eine (oder mehre) Produktionsregel(n) mehrfach auf einen Graphen angewendet – solange bis keine Reduktion mehr möglich ist – wird von einer Transformation gesprochen. Sofern es dabei in einem Schritt mehrere Alternativen für eine weitere Reduktion gibt, wird genau eine davon weiterverfolgt – gemäß einer vorher definierten Redexauswahlfunktion[4]. Werden andererseits alle Alternativen gleichzeitig weiterverfolgt, kann die Menge aller so alternativ entstehenden Graphen als eine Sprache verstanden werden. Die Produktionsregel(n) werden, zusammen mit dem ursprünglichen Graphen, als eine Graphgrammatik bezeichnet (vgl. Abbildung 2.5).

	Graph g	Produktionsregeln P = { p1,...,pN }	Verhalten bei mehreren Redexen als Kandidaten für die nächste Reduktion
Graphgrammatik: Die Menge aller erreichbarer Graphen ist genau die Menge aller Wörter der zu beschreibenden Sprache	fest (Startgraph)	fest	alle Redexe verfolgen & Ergebnisse aufsummieren
Transformation: Als Ergebnis entsteht genau ein Graph - in Abhängigkeit der Wahl des zu Transformierenden Graphs g	variabel (zu transformierender Graph)	fest	nur ein Redex verfolgen, gemäß einer Redexauswahlfunktion

Abbildung 2.5 Zusammenhang zwischen Graphgrammatiken und Transformationen

Definition 2.29: Graphgrammatik
*Eine **Graphgrammatik*** gg ∈ GG *ist ein Tupel* (s,P) *wobei* s *ein Graph und* P *eine Menge von Produktionsregeln ist:*

$$GG =_{def} GRAPH \times Pot(PRODRULE)$$

Der Startgraph s wird auch als Axiom bezeichnet (vgl. [35]).

Definition 2.30: Extension einer Graphgrammatik
*Die **Extension einer Graphgrammatik*** gg *wird mit* Ext(gg) *notiert und ist wie folgt definiert:*

$$Ext(\ (s,P)\ ) =_{def} \{\ s\ \} \cup [\ \bigcup p \in P: [\ \bigcup hx \in pmf(p,s): [\ \bigcup iso \in \{f : DB(hc)/DB(hx) \rightarrow GE/\{s\}\} : Ext(iso(g),P)\ ]\ wobei\ (g,hc) = reduce(s,p,hx)\ ]\ ]$$

Zur Extension gehört der Startgraph s sowie alle Graphen, die sich durch Anwendung einer beliebigen Produktionsregel p auf einen beliebigen Redex in s ergeben. Jeder der neu erzeugten Graphen wird dann im Rahmen der Rekursion seinerseits als ein Startgraph betrachtet, wodurch auch alle dessen möglichen Veränderungen zur Extension gehören. Eine letzte Dimension für mögliche Elemente der Extension wird durch die Vergabe der Ids aufgestellt. Da eine Reduktion jeweils neue Ids vergibt, würden so, gemäß der Definition der Funktion newGE(), bereits existierende Graphen nicht zur Extension gehören können, da deren Ids im Rahmen der Berechnung hier nicht auftauchen würden. Weiterhin ist zu berücksichtigen, dass eine bestimmte Id an einer beliebigen Stelle eines Graphen der Extension stehen kann. Entsprechend werden für ein Reduktionsergebnis g alle Isomorphien für die durch den Ersetzungsmorphismus hc neu erzeugte Graphelemente betrachtet. Jede solche Isomorphie iso wird auf g angewendet und bildet letztendlich ein Element der

[4] In der Funktionalen Programmierung (vgl. [34]) auch als Reduktionsstrategie bezeichnet.

Extension. Als Ids kommen alle die in Frage, die nur noch nicht im (jeweils aktuellen) Stargraph s verwendet wurden.

Die Berechnung der Extension kann nicht-terminierend sein, wodurch unendlich große Sprachen erst beschrieben werden können. Als Produktionsregeln können daher auch solche verwendet werden, die kein Löschmuster besitzen. Damit kann auf einfache Weise beschrieben werden, dass eine bestimmte Struktur von Graphelementen beliebig häufig in den Wörtern einer Sprache auftreten darf. Abbildung 2.6 zeigt dazu ein Beispiel einer Graphgrammatik gg, mit der die Sprache aller „doppelt verketteter Bäume“ beschrieben ist.

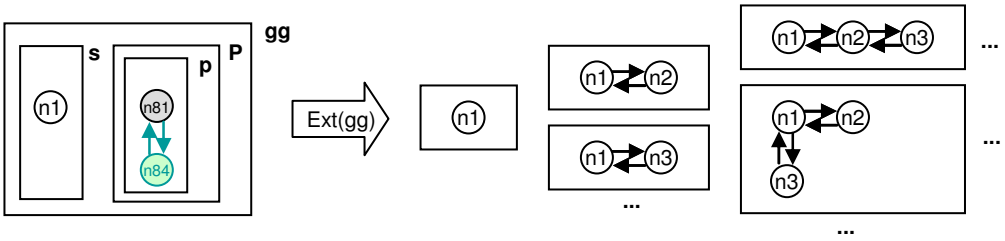


Abbildung 2.6 Extension einer Graphgrammatik

Definition 2.31: Redexauswahlfunktion
Für eine Produktionsregel p ∈ PRODRULE *und einen Graphen* g ∈ GRAPH *kann die Menge der Redexmorphismen mehr als ein Element enthalten. Wird eine Menge von Produktionsregeln* P *verwendet, erweitert sich die Menge der Redexmorphismen entsprechend auf die Vereinigung der Redexmorphismen aller* p ∈ P. *Die* ***Redexauswahlfunktion*** rsf *ist eine Funktion, die aus einer solchen Menge genau einen Redexmorphismus auswählt, und wie folgt spezifiziert ist:*

$$\forall K \in Pot(PRODRULE \times REDEXMORPH) : rsf(K) =_{spec} any(K)$$

Eine konkrete Berechnungsvorschrift für diese Funktion ist für das Weitere in diesem Grundlagenkapitel nicht relevant (für mögliche Ausgestaltungen sei beispielhaft auf das Lambda-Kalkül [36] verwiesen). Aus einer Menge K von „Reduktionskandidaten“ sucht die Redexauswahlfunktion eine aus. Ein Reduktionskandidat besteht hierbei nicht nur aus einem Redexmorphismus, sondern auch aus der dazugehörigen Produktionsregel, auf den sich der Redexmorphismus bezieht. Die explizite Angabe der Produktionsregel ist notwendig, da sie im Allgemeinen nicht anhand des Redexmorphismus eindeutig bestimmt werden kann, für die Ausführung der Reduktion mit dem gewählten Redex jedoch gebraucht wird.

Definition 2.32: Transformation
Die Menge TRANS *aller* ***Transformationen*** *ist wie folgt definiert:*

$$TRANS =_{def} Pot(PRODRULE)$$

Eine Transformation ist eine Menge von Produktionsregeln.

Definition 2.33: Transformationsanwendung
*Die **Anwendung einer Transformation*** tr ∈ TRANS *auf einen Graphen* g ∈ GRAPH*, notiert als* trans(g,tr)*, ist wie folgt definiert:*

$$
\begin{aligned}
&trans(g,tr) =_{def} \text{falls } |K|=0 \text{ dann } g \text{ sonst } trans(\, g'', tr\,) \\
&\text{wobei} \\
&\quad K = [\bigcup p \in tr: \{p\} \times pmf(p,g)] \\
&\quad (p,hx) = rsf(K) \\
&\quad (g'',hc) = reduce(g,p,hx)
\end{aligned}
$$

Es wird solange reduziert bis keine Redexe mehr vorhanden sind. Beim Aufstellen der Produktionsregeln ist daher darauf zu achten, dass die Transformation terminiert, um praktisch anwendbar zu sein. Dies schließt jedoch nicht zwingend solche Produktionsregeln aus, die für sich genommen nicht terminierend angewendet werden können (wie z.B. Produktionsregeln effektiv keine Graphelemente löschen). In Analogie zur Umsetzung der Rekursion im Lambda-Kalkül [36] kann eine Terminierung auch durch eine andere Produktionsregel (die die Redex bildenden Graphelemente entfernt) in Verbindung mit einer geeigneten Redexauswahlfunktion herbeigeführt werden.

Im Folgenden wird unter dem Begriff Transformation je nach Kontext sowohl die Menge der Produktionsregeln als auch deren Anwendung verstanden.

2.3 Aktualisieren einer Transformationsanwendung

In der Praxis können Transformationen für verschiedene Zwecke eingesetzt werden. Beispielsweise kann eine Datei in ein anderes Dateiformat konvertiert, oder aus einem UML-Diagramm Quelltext generiert werden (vgl. [37]). Bei der Anwendung einer Transformation wird die Eingabe nicht notwendigerweise „zerstört“, sondern kann (sofern nur eine Kopie transformiert wurde) danach sogar noch weiter bearbeitet werden. Das ist besonders im Rahmen des letzten oben genannten Beispiels relevant, wenn nach der ersten Transformation nicht nur die ursprüngliche Eingabe, sondern auch die Ausgabe weiterentwickelt wird. Abbildung 2.7 zeigt das dabei entstehende Problem anhand eines ursprünglichen Graphen g0 und der Transformation tr, die aus genau der einen Produktionsregel p besteht. Zum Zeitpunkt t1 wird Graph g1 durch Anwenden von tr auf (eine Kopie von) g0 erzeugt. Anschließend wird g1 per Hand um den Knoten n80 und die Kante e80 erweitert, die den Knoten e80 an den Knoten n10 bindet. Es entsteht g1'. Zum Zeitpunkt t3 (wahlweise auch gleichzeitig zum vorherigen Schritt) wird g0 per Hand um die Kante e9 erweitert. Es entsteht g0'. Da durch e9 nun ein weiterer Redex existiert (und damit zu einem anderen Transformationsergebnis führt), soll nun zum Zeitpunkt t4 mit tr erneut transformiert werden. Es entsteht g1''. Dieser Graph enthält jedoch nicht die Erweiterung die zum Zeitpunkt t2 vorgenommen wurden (Graphelemente n80 und e80). Da zudem neue Graphelement-Ids generiert wurden (n20 statt n10, e20 statt e20 usw.), ist eine automatische Übertragung der in t2 hinzugefügten Elemente nicht eindeutig möglich.

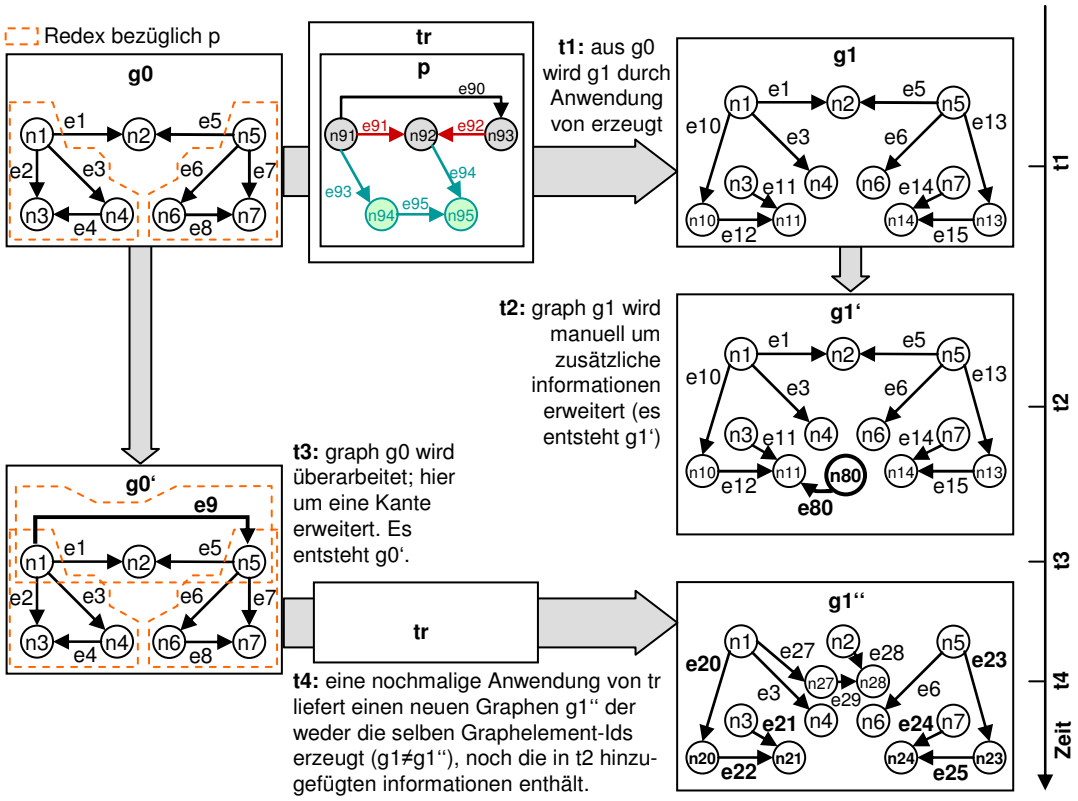


Abbildung 2.7 Motivation zur Aktualisierung von Transformationen

Unter einer Aktualisierung einer Transformationsanwendung wird der Vorgang bezeichnet, bei dem eine erneute Transformationsanwendung vorgenommen wird, wobei gleichzeitig die in der Zwischenzeit zur Ausgabe manuell hinzugefügten Informationen in die neue Ausgabe *korrekt* übertragen werden. Die Kapitel 2.3.1 stellt ein entsprechendes Verfahren vor. In dem Kapitel 2.3.2 werden die Grenzen der Aktualisierung von Transformationen erörtert.

2.3.1 Inkrementelle Transformation

Die inkrementellen Transformation zur Aktualisierung einer Transformationsanwendung gestaltet sich wie folgt: Das Beispiel aus Abbildung 2.7 aufgreifend, wird bei der aller ersten Transformationsanwendung auf einen Graphen g0 jede Reduktion durch einen Regelmorphismus dokumentiert, d.h. abgespeichert. Bei der nächsten Transformationsanwendung, diesmal mit g0' als Eingabe, wird für jeden gefundenen Redexmorphismus geprüft, ob dieser bereits in einer früheren Transformation verarbeitet wurde. Falls keiner der abgespeicherten Regelmorphismen einen solchen Redexmorphismus hat, wird mit dem neu gefundenen Redexmorphismus reduziert, und der so entstandene Regelmorphismus zu den bisherigen dazugelegt. Falls jedoch ein Regelmorphismus mit dem gleichen Redexmorphismus gefunden wurde, wird anstelle einer neuen Reduktion der abspeicherte Regelmorphismus erneut angewendet. So können genau dieselben Ids wie bei der ersten Transformationsanwendung „erzeugt" werden. Anschließend können die zwischen den Transformationsanwendungen manuell zum Transformationsergebnis hinzugefügten Graphelemente (zum Zeitpunkt t2) übertragen werden, da die referenzierten Graphelemente ihre Ids „beibehalten" (oder ganz gelöscht werden).

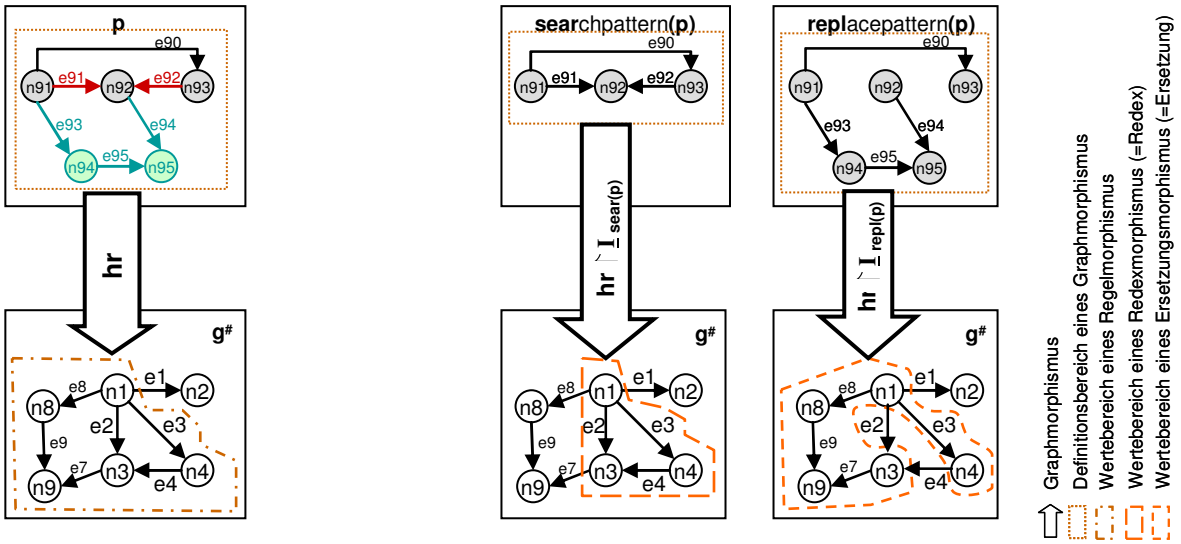
Definition 2.34: Regelmorphismus, Redexmorphismus, Ersetzungsmorphismus
Gegeben sei ein Graph $g^{\#} \in$ GRAPH *und eine Produktionsregel* p. *Ein Graphhomomorphismus* hr *bezüglich* p *und* $g^{\#}$*, der also das gesamte Muster von* p *auf (einen Teil von)* $g^{\#}$ *abbildet, wird als* ***Regelmorphismus*** *bezüglich* p *und* $g^{\#}$ *bezeichnet. Die Menge* RULEMORPH *aller Regelmorphismen ist wie folgt definiert:*

$$\text{RULEMORPH} =_{def} \bigcup_{p \in \text{PRODRULE}}, \bigcup_{g^{\#} \in \text{GRAPH}}: \{ hr \in \text{GRAPHMORPH} \mid \text{homomorph}<hr,p,g^{\#}> \}$$

Der ***Redexmorphismus*** *bzw.* ***Ersetzungsmorphismus*** *eines Regelmorphismus* hr *bezüglich einer Produktionsregel* p*, notiert als* redexmorph(hr,p) *bzw.* replacemorph(hr,p)*, ist wie folgt definiert:*

$$\text{redexmorph}(hr,p) =_{def} hr \restriction \underline{Id}_{\text{searchpattern}(p)}$$
$$\text{replacemorph}(hr,p) =_{def} hr \restriction \underline{Id}_{\text{replacepattern}(p)}$$

Ein Regelmorphismus hr dokumentiert eine *geschehene* Anwendung einer Produktionsregel p auf einen Graphen g, der zu g' verändert wurde. Um sowohl gelöschte als auch erzeugte Graphelemente zu dokumentieren, bildet hr als Graphmorphismus nicht nur auf g oder g', sondern auf deren Vereinigung ab, die hier als $g^{\#}$ geführt wird (mit $g^{\#} = g \cup g'$). Aus hr lässt sich die durchgeführte Ersetzung mit der Beschränkung auf das Such- und Ersetzungsmuster ableiten (vgl. Abbildung 2.8). Der Redexmorphismus beschreibt, *wie* die Produktionsregel p auf einen Teil in g abgebildet wurde. Der Ersetzungsmeorphismus beschreibt, wie jener Teil anhand von p ersetzt wurde.



hr = { n91↦n1,n92↦n3,n93↦n4,n94↦n8,n95↦n9, e90↦e3,e91↦e2,e92↦e4,e93↦e8,e94↦e7,e95↦e9 }
hr↾ $I_{sear(p)}$ = { n91↦n1,n92↦n3,n93↦n4, e90↦e3,e91↦e2,e92↦e4 }
hr↿ $I_{repl(p)}$ = { n91↦n1,n92↦n3,n93↦n4,n94↦n8,n95↦n9, 90↦e3, e93↦e8,e94↦e7,e95↦e9 }

Abbildung 2.8 Durch einen Regelmorphismus dokumentierte Informationen

Definition 2.35: Transformationshistorie
Die Menge TRANSHISTORY *der* ***Transformationshistorien*** *ist wie folgt definiert:*

TRANSHISTORY $=_{def}$ { c ⊂ PRODRULE×RULEMORPH |
[∀ p∈PRODRULE, hx∈REDEXMORPH: |{ (p,hr)∈c | hx⊆hr }|≤1] }

Für eine Transformationshistorie c ∈ TRANSHISTORY*, eine Produktionsregel* p ∈ PRODRULE *und ein Redexmorphismus* hx ∈ REDEXMORPH *liefert die Funktion* rulemorph *denjenigen Regelmorphismus aus* c*, der* hx *als Redexmorphismus hat und sich auf* p *bezieht*:

rulemorph(c,p,hx) $=_{def}$ any({ hr ∈ RULEMORPH | (p,hr)∈c, redexmorph(hr,p)=hx })

Eine Transformationshistorie dokumentiert geschehene Anwendungen von Produktionsregeln. Da bei mehreren Produktionsregeln in einer Transformation sich die jeweils angewendete nicht eindeutig aus einem Regelmorphismus bestimmen lässt, ist ein Eintrag in der Transformationshistorie ein Paar aus Produktionsregel und Regelmorphismus. Pro Produktionsregel p und Redexmorphismus hx darf es jedoch maximal einen Eintrag geben (bei dem der Regelmorphismus hx als dessen Redexmorphismus hat). Das Beispiel aus Abbildung 2.7 fortführend, zeigt Abbildung 2.9 die Transformationshistorie c1, der durch die Ausführung der Transformation tr auf g0 zum Zeitpunkt t1 entsteht. Er enthält zwei Einträge die sich beide auf die Produktionsregel p beziehen. Der erste Eintrag resultiert aus der Reduktion mit dem Redexmorphismus hx1, der zweite durch Reduktion mit hx2.

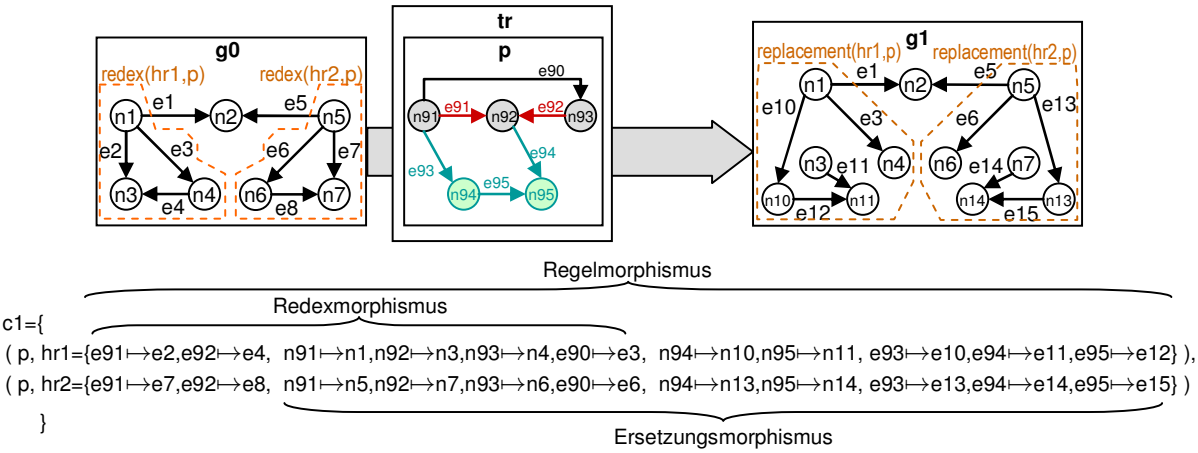


Abbildung 2.9 Transformationshistorie

Definition 2.36: Rereduktion
Gegeben sei ein Graph g ∈ GRAPH, *eine Produktionsregel* p ∈ PRODRULE *und ein Regelmorphismus* hr ∈ RULEMORPH *bezüglich* g *und* p. *Die* ***Rereduktion*** *des Graphen* g *bezüglich* hr, *notiert als* rereduce(g,p,hr), *ist wie folgt definiert:*

rereduce(g,p,hr) $=_{def}$ [g – (redex/replacement)] + (replacement/redex) mit
redex = [redexmorph(hr,p)](searchpattern(p))
replacement = [replacemorph(hr,p)](replacepattern(p))

Eine Rereduktion ist wie eine Reduktion mit dem Unterschied, dass anstelle des Einsatzes einer Erzeugungsfunktion, der im Regelmorphismus hr bereits existierende Ersetzungsmorphismus verwendet wird. So können frühere Reduktionen (sofern sie in einer Transformationshistorie abgespeichert wurden) wiederholt ausgeführt werden – ggf. auf einen sich in der Zwischenzeit geänderten Ausgangsgraphen (vgl. g0 zu g0' in Abbildung 2.7).

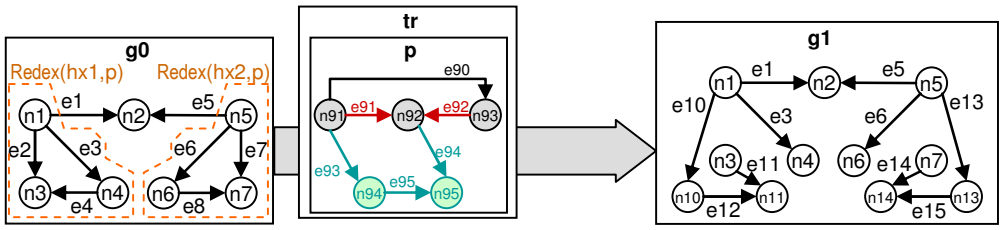
Definition 2.37: Inkrementelle Transformationsanwendung
Gegeben sei eine Transformation tr ∈ TRANS, *ein Graph* g0 ∈ GRAPH *und eine Transformationshistorie* c0 ∈ TRANSHISTORY. *Die* ***inkrementelle Transformationsanwendung*** *bezüglich* g0, tr *und* c0, *notiert als* inktrans(g0,tr,c0), *ist wie folgt definiert:*

inktrans(g0,tr,c0) $=_{def}$ inktrans(g0,tr,c0,∅)

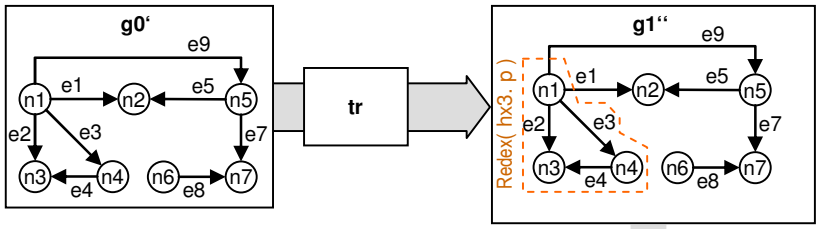
inktrans(g,tr,c0,c1) $=_{def}$ falls |K|=0 dann (g,c1) sonst inktrans(g'', tr, c0, c1'')
wobei
K = [$\bigcup_{p \in tr}$: {p}×pmf(p,g)]
(p,hx) = rsf(K)
[falls rulemorph(c0,p,hx) def. dann
g'' = rereduce(g,p,rulemorph(c0,p,hx))
hc = replacemorph(rulemorph(c0,p,hx))
sonst
(g'',hc) = reduce(g,p,hx), einmalig ausgeführt]
c1'' = c1∪{(p,hx∪hc)}

Bei der inkrementellen Berechnung des Transformationsergebnisses wird die bisherige Transformationshistorie c0 separat von der neu erstellten c1 gehalten, so dass in c1 nur solche Regelmorphismen enthalten sind, die während der Berechnung auch tatsächlich ausgeführt wurden. Insgesamt ist die Berechnung tailrekursiv organisert, wobei in jedem Schritt genau eine (Re-)Reduktion vorgenommen wird, und das Ende erreicht ist, sobald keine

Redexmorphismen mehr vorhanden sind (Menge K). Liegt in einem Schritt nach Ausführung der Reduktionsstrategiefunktion rsf ein Redexmorphismus hx mit zugehöriger Produktionsregel p vor, ergibt sich das Weitere aus der Frage, ob hx in der bisherigen Transformationshistorie c0 enthalten ist. Wenn das der Fall ist, wird mit dem in c0 enthaltenen Regelmorphismus reduziert (der hx als Redexmorphismus hat, und sich auf die Produktionsregel p bezieht). Andernfalls wird eine Reduktion wie bei einer normalen Transformation durchgeführt und der dabei erstellte Ersetzungsmorphismus, zusammen mit hx und p, als neuer Eintrag in der Transformationshistorie abgelegt. Abbildung 2.10 veranschaulicht dies mit zwei Berechnungen für das in Abbildung 2.7 eingeführte Beispiel.

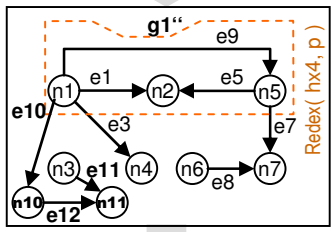


c0 ={}
hx1={ n91↦n1,n92↦n3,n93↦n4,e90↦e3,e91↦e2,e92↦e4 }
hx2={ n91↦n5,n92↦n7,n93↦n6,e90↦e6,e91↦e7,e92↦e8 }
c1={
hr1=(p, {n91↦n1,n92↦n3,n93↦n4,e90↦e3,e91↦e2,e92↦e4, n94↦n10,n95↦n11, e93↦e10,e94↦e11,e95↦e12}),
hr2=(p, {n91↦n5,n92↦n7,n93↦n6,e90↦e6,e91↦e7,e92↦e8, n94↦n13,n95↦n14, e93↦e13,e94↦e14,e95↦e15})
}

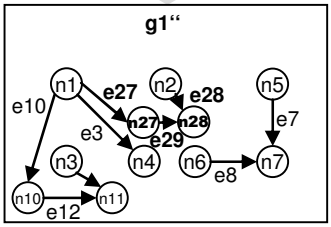


vor 1.Reduktion

hx3={ n91↦n1,n92↦n3,n93↦n4,e90↦e3,e91↦e2,e92↦e4 }
da der Redexmorphismus von hr1 (in c1) gleich hx3 ist, wird g'' mit (p,hr1) rereduziert,
und (p,hr1) in die neue Transformationshistorie aufgenommen.



hx4={ n91↦n1,n92↦n2,n93↦n5,e90↦e9,e91↦e2,e92↦e5 }
da hx4 in keinem Regelmorphismus in c0 vorkommt, wird g'' mit (p,hx4) reduziert,
resultierender Regelmorphismus wird in neuer Transformationshistorie aufgenommen



Ende
da keine weiteren
Redexe vorhanden

c1={
(p, {n91↦n1,n92↦n3,n93↦n4,e90↦e3,e91↦e2,e92↦e4, n94↦n10,n95↦n11, e93↦e10,e94↦e11,e95↦e12}),
(p, {n91↦n1,n92↦n2,n93↦n5,e90↦e9,e91↦e2,e92↦e5, n94↦n27,n95↦n28, e93↦e27,e94↦e28,e95↦e29}) }

Abbildung 2.10 Inkrementelle Berechnung eines Transformationsergebnisses

Die erste Berechnung in Abbildung 2.10 nimmt die leere Transformationshistorie c0 als Eingabe, und liefert die Transformationshistorie c1 als Ausgabe, in dem zwei Einträge für die

beiden Redexmorphismen hx1 und hx2 enthalten sind. Danach wurde das transformierte g0 durch löschen der Kante e6 zu g0' verändert. Bei der anschließenden zweiten Berechnung wird c1 als Eingabe, und zu c1'' als Ausgabe geliefert. Dabei werden die beiden Redexmorphismen hx3 und hx4 gefunden. Da hx3 dem Redexmorphismus hr1 des ersten Regelmorphismus in c1 gleicht, wird mit hr1 rereduziert. Da hx4 nicht in c1 vorkommt, wird reduziert.

Definition 2.38: Szenario der inkrementellen Transformation
Geben sei eine Transformation tr ∈ TRANS *sowie eine referenziell nicht transparente Funktion* user : GRAPH → GRAPH *die das Verhalten eines Anwenders bei der Erstellung eines neuen bzw. Veränderung eines gegebenen Graphen beschreibt. Das* ***Szenario der inkrementellen Transformationsanwendung*** *bezüglich* tr *und* user*, notiert als* $scenink_{user}(tr)$*, ist wie folgt definiert:*

$scenink_{user}$(tr) =$_{def}$ scenink(g0, tr, g1, c1)
wobei
  g0 = user(∅)
  (g1,c1) = inktrans(g0, tr, ∅)

$scenink_{user}$(g0, tr, g1, c1) =$_{def}$ $scenink_{user}$(g0', tr, g1''', c1')
wobei
  g0' = user(g0)
  g1' = user(g1)
  (g1'',c1') = inktrans(g0', tr, c1)
  g1''' = g1'' + (g1'–g1) – (g1–g1')

Der erste Definitionsteil beschreibt die Ausgangssituation, wie sie in Abbildung 2.7 (oben) gezeigt ist: Ein Anwender erstellt einen ersten Graphen g0 (hier formal durch Veränderung eines leeren Graphen), der mit tr und einer leeren Transformationshistorie zum Graph g1 und der neuen Transformationshistorie c1 transformiert wird. Der zweite Definitionsteil beschreibt jeden Schritt des weiteren Ablaufs (Hauptteil in Abbildung 2.7): Der Anwender verändert parallel g0 und g1, wodurch g0' und g1' entstehen. Anschließend wird g0' mit tr und c1 zu g1'' und c1' transformiert. Um die zwischen dieser Transformationsanwendungen und der vorherigen hinzugefügten Graphelemente (bzw. entfernte) auf das aktualisierte Transformationsergebnis g1'' zu übertragen, müssen diese zunächst ermittelt werden. Dazu werden die Differenzen aus g1' und g1 (vgl. Abbildung 2.7) gebildet[5]. Diese Differenzen werden anschließend auf g1'' angewendet, wodurch g1''' entsteht.

2.3.2 Konflikt

Ein Konflikt liegt vor, wenn ein Transformationsergebnis nicht wohlgeformt ist. Dies geschieht, wenn die zu einem Transformationsergebnis hinzugefügten Kanten auf Knoten verweisen die im aktualisierten Transformationsergebnis nicht mehr vorhanden sind. Abbildung 2.11 zeigt dies an einem Beispiel: Aus dem ursprünglichen Graphen g0 wird zum Zeitpunkt z1 durch Anwendung der Transformation tr das Transformationsergebnis g erzeugt. Dieses wird um die Graphelemente n80 und e80 erweitert wodurch g' entsteht. Nach einer Veränderung von g0 zu g0' (durch Löschung der Kante e3) und einer Aktualisierung der Transformationsanwendung zeigt im Ergebnis g'' die Kante e80 auf einen nicht mehr vorhandenen Knoten.

[5] Sofern g1 nicht mehr verfügbar ist, kann dieser auch aus g0, tr und c0 durch eine nochmalige Transformation ermittelt werden. Alternativ können die Differenzen aus g1 und g1' auch explizit in Form eines Editscripts vorliegen (vgl. [38]) – so dass weder g0 noch g1 noch verfügbar sein müssen.

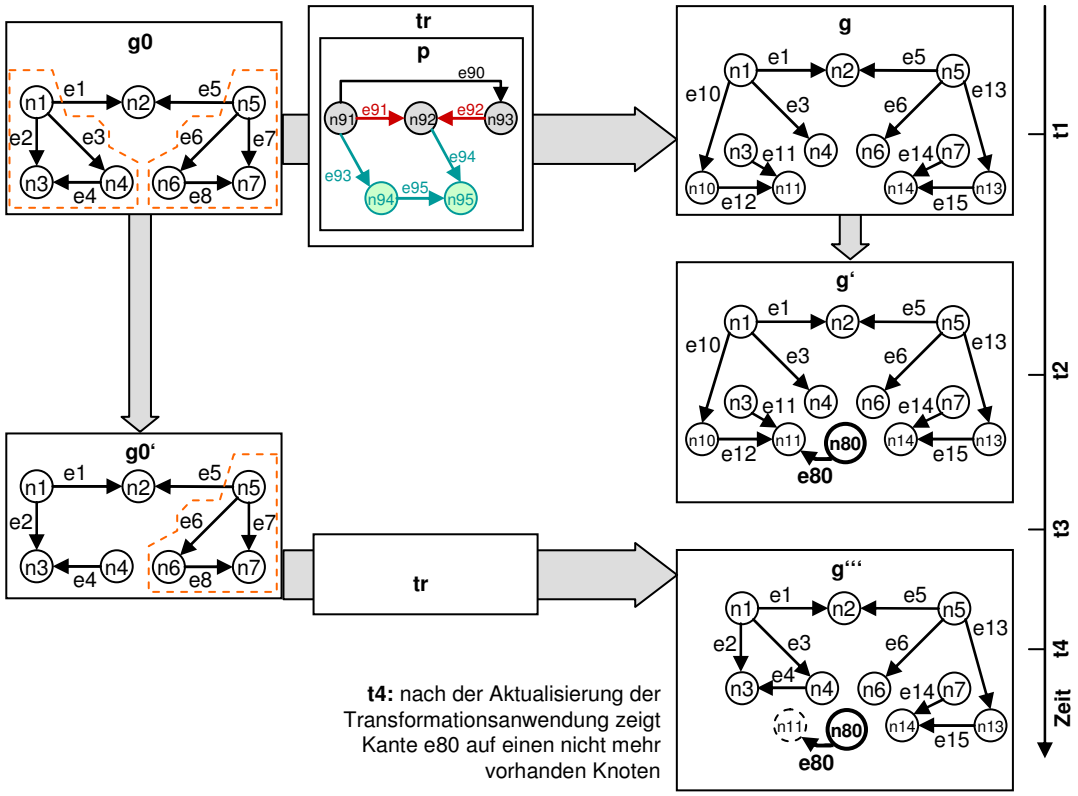


Abbildung 2.11 Konfliktsituation: manuell hinzugefügte Elemente verlieren ihren Bezug

Ein Konflikt kann auf verschiedene Weisen aufgelöst werden: So kann beispielsweise die Kante e80 entfernt werden, oder auf ein anderes Element, z.B. n14, verweisen. Welche Korrekturmaßnahme die „richtige“ ist, lässt sich jedoch nicht anhand der hier vorliegenden Informationen entscheiden. Im Allgemeinen kann dies sogar von der Intention des Hinzufügers der Kante abhängen, die wiederum ggf. formal nicht greifbar ist.

Definition 2.39: Szenario der inkrementellen Transformation mit Konfliktbehebung
Geben sei eine (ggf. referenziell nicht transparente) Funktion resolveconflict : GRAPH → GRAPH *die die Auflösung eines Konflikts in einem Graphen beschreibt. Für den Einsatz dieser Funktion im Szenario der inkrementellen Transformationsanwendung bezüglich einer Transformation* tr *ist folgende Anpassung (fett markiert) vorzunehmen:*

$scenink_{user}$(g0, tr, g1, c1) $=_{def}$ $scenink_{user}$(g0', tr, g1‘‘‘, c1')
wobei
g0' = user(g0)
g1' = user(g1)
(g1'',c1') = inktrans(g0', tr, c1)
g1‘‘‘ = **resolveconflict(** g1'' + (g1'–g1) – (g1–g1') **)**

Die Konfliktauflösungsfunktion resolvekonflikt arbeitet hier nur auf dem um die manuellen Änderungen angereicherten Transformationsergebnis g1‘‘‘. Dies ist jedoch nicht als Beschränkung der Allgemeinheit zu verstehen. So könnte eine Funktion zur Konfliktlösung als weitere Parameter die vorgenommenen Änderungen (Differenzen aus g1' und g), die ursprünglichen Graphen g0 und g1, sowie auch die gesamte Historie aller bisher durchgeführten Transformationen und Konfliktauflösungen entgegennehmen. Damit ist in dieser Arbeit die Schnittstelle für geeignete Konfliktbehandlungen definiert – ohne auf diese im Weiteren im Detail einzugehen.

2.4 Erweiterungen für Graphen, Grammatiken, Transformationen

In diesem Kapitel werden Graphen, Graphgrammatiken und Transformationen um Konzepte erweitert, die deren Anwendbarkeit steigern. Das jeweilige Konzept wird dabei als Differenz zu den bisherigen Grundlagen dargestellt. Für Strukturen werden neue Bestandteile definiert (bei Graphen beispielsweise ein Bestandteil für Typzuordnungen), für Funktionen werden Anpassungen definiert, die die bisherigen Funktionsdefinitionen „ersetzen“. Abbildung 2.12 zeigt dazu ein Überblick über die Wirkung solcher Anpassungen: Die bisher definierten Funktionen bauen aufeinander auf und bilden dadurch ein Abhängigkeitsnetz. Wird beispielsweise die Funktion reduce geändert, so wirkt sich dies implizit auch auf die Funktionen Ext, trans und inktrans aus – ohne dass dazu Anpassungen explizit definiert werden müssten. Um innerhalb einer Definition zwischen der ursprünglichen und der angepassten Fassung einer Funktion zu unterscheiden, wird letztere mit einem ° markiert.

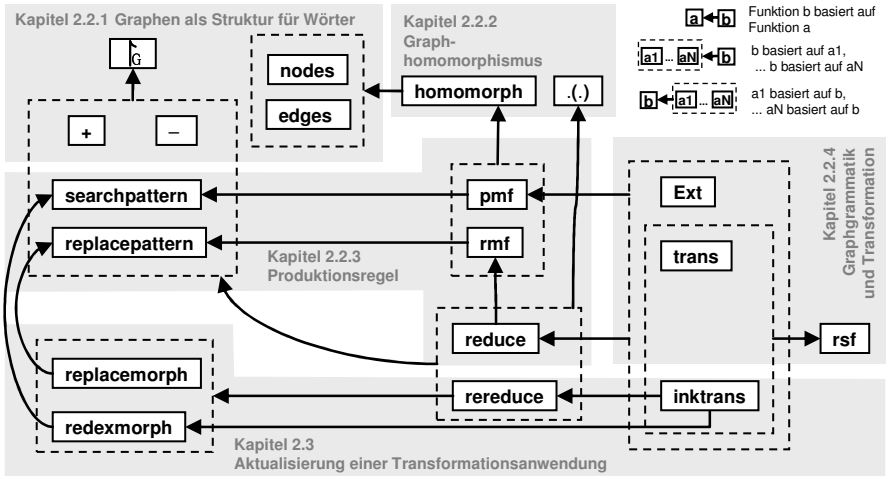


Abbildung 2.12 Abhängigkeitsnetz als Grundlage zur Beschreibung erweiternder Konzepte

2.4.1 Domäne

Bei der bisher definierten Art der Transformationsausführung entstand der Ausgabegraph durch Reduktionen aus dem Eingabegraph. Dabei wurden im Allgemeinen nach und nach Graphelemente des Eingabegraphen durch neue, gemäß der Produktionsregeln erzeugte, ersetzt. Anders formuliert waren Ein- und Ausgabe nicht in zwei Graphen getrennt – sondern beide teilten sich einen Graphen wobei die Ausgabe durch „Mutation“ aus der Eingabe entstand. Die Nutzung desselben Graphen für Ein- und Ausgabe hat jedoch zur Folge, dass neu erzeugte Elemente auch uneingeschränkt als Kandidaten für die Belegung der Suchmuster der Produktionsregeln in Frage kommen. Abbildung 2.13 zeigt in der oberen Hälfte wie nach der ersten Anwendung der Produktionsregel p1 der erzeugte Knoten n11 selbst einen neuen Redex bildet, dadurch wieder entfernt wird, und so schließlich Knoten n12 entsteht.

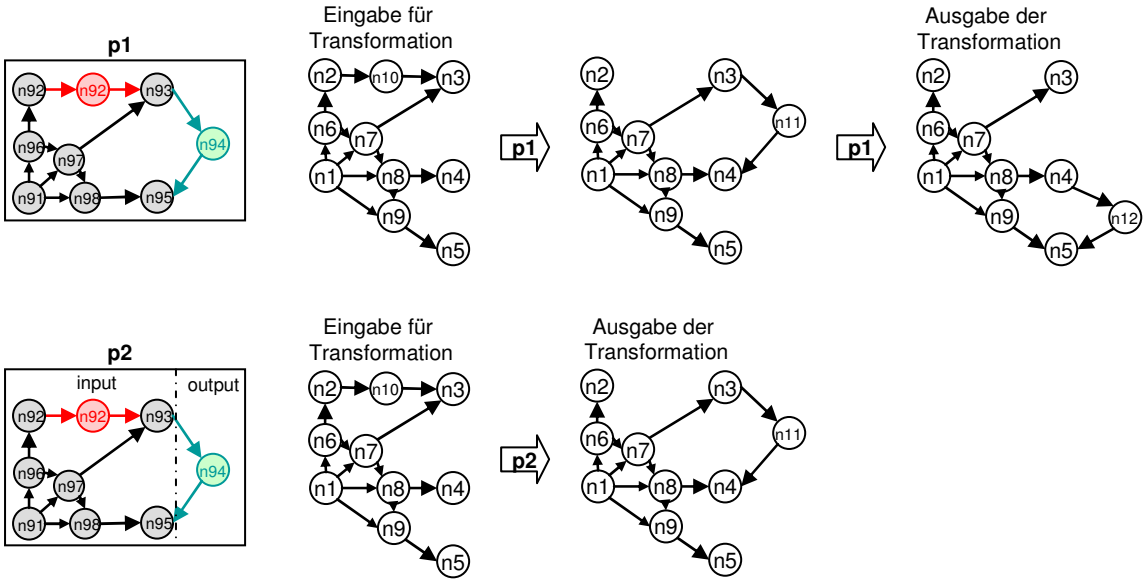
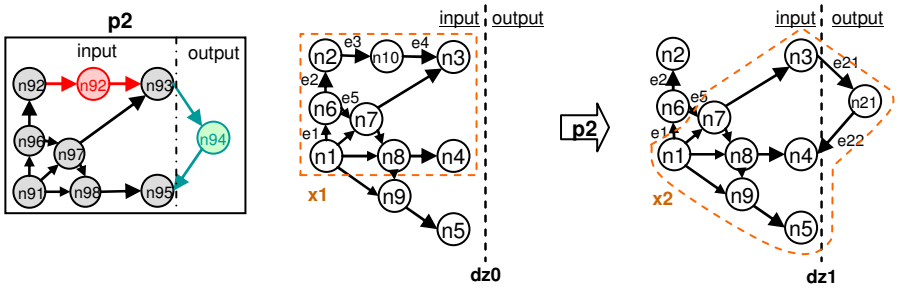


Abbildung 2.13 Einsatz von Domänen zur einfacheren Definition von Produktionsregeln

Dieses Verhalten der Transformationsausführung ist nicht immer gewünscht. Bei der Definition einer Transformation muss dann darauf geachtet werden, ob in den erzeugten Strukturen sich unbeabsichtigte Redexe bilden können. Das ist im Allgemeinen nicht auf Anhieb leicht festzustellen, da dazu alle Kombinationen von Produktionsregelanwendungen durchzuspielen sind. Sofern Kollisionen festgestellt werden, müssen die Produktionsregeln entsprechend umgebaut und erneut geprüft werden.

Ein einfacheres Mittel zur Umgebung dieses Problems ist der Einsatz von Domänen. In Abbildung 2.13 Einsatz von Domänen zur einfacheren Definition von Produktionsregeln (untere Hälfte) ist die Produktionsregel p2 mit den zwei Domänen input und output dargestellt. Die Transformationsausführung wird dahingehend angepasst, dass Graphelemente, die „innerhalb“ der output-Domäne erzeugt wurden, nicht für Redexe für die input-Domäne herangezogen werden. Abbildung 2.14 zeigt wie dies im Detail funktioniert: Eine zu jedem Graph mitgeführte Domänenzuordnung (z.B. dz0) ordnet dessen Elemente den Domänen input und output zu. Ein Redex wird nur dann als gültig betrachtet, wenn durch den entsprechenden Redexmorphismus jedes Graphelement des Suchmusters und dessen Belegung in derselben Domäne liegen. Während dies vor der ersten Anwendung der Produktionsregel p2 der Fall ist (Redex x1), liegt im resultierenden Graph die Belegung von n92 in einer anderen Domäne als der Knoten n21 was diesen Redex (x2) von einer weiteren Reduktion ausschließt.



dz0 = { input ↦ {n1,n2,n3, ..., n9, n10, e1,e2,e3,e4,e5,...}, output ↦ { } }
dz1 = { input ↦ {n1,n2,n3, ..., n9, e1,e2, ,e5,...}, output ↦ {n21,e21,e22} }

Abbildung 2.14 Filtern von Redexen anhand der Domänenzuordnung

Zu beachten ist, dass der Zuschnitt der Domänen unabhängig der Wahl der add- und del-Markierungen sind. So können in allen Domänen sowohl Löschungen als auch

Hinzufügungen stattfinden. Jede dieser Domäne kann folglich selbst als eine eigenständige Produktionsregel betrachtet werden – mit dem Zusatz, dass Links über Graphgrenzen hinweg gehen dürfen. Dies hat auch keine Auswirkungen auf die Ausführung einer inkrementellen Transformation, da der einzige Unterschied in der Auswahl der Redexmorphismen besteht – was orthogonal zu deren Abspeichern und Vergleichen ist.

Definition 2.40: Domäne
*Eine **Domäne*** $d \in DOMAIN$ *ist ein Index für eine Graphelementmenge. Im Rahmen dieser Arbeit erfüllt diese Menge die folgende Spezifikation:*

$$DOMAIN \supset_{spec} \{ input, output, ld, cm, rd \}$$

Die Domänen input und output wurden bereits in Abbildung 2.14 verwendet. Die übrigen Domänen werden an gegebener Stelle erläutert. Im weiteren Sprachgebrauch wird mit einer Domäne auch die durch sie in einer Graphpartition (siehe nachfolgende Definition) referenzierten Menge von Graphelementen bezeichnet.

Definition 2.41: Graphpartition
*Eine **Graphpartition*** $g \in GRAPHPART$ *teilt eine Graphelementmenge in eine Menge von indexierten und zueinander disjunkten Graphelementmengen auf:*

$$GRAPHPART =_{def} \{ gp \mid gp: DOMAIN \rightarrow Pot(GE) \wedge [\forall d1,d2 \in DB(gp): gp(d1) \cap gp(d2) = \emptyset] \}$$

Definition 2.42: Graph mit Domänen
Für das Konzept der Domänen ist die Definition der Menge GRAPH *wie folgt anzupassen:*

$$GRAPH^{\circ} =_{def} \{ g \in GRAPH \mid \underline{dom}_g \in GRAPHPART \}$$

Jeder Graph enthält zusätzlich zu seinen bisherigen Bestandteilen den Bestandteil $\underline{dom}$, der eine Graphpartition ist und die Graphelemente des Graphen disjunkt auf Domänen aufteilt.

Definition 2.43: Patternmatchingfunktion mit Domänen
Für das Konzept der Domänen ist die Definition der Patternmatchingfunktion pmf *wie folgt anzupassen:*

$$pmf^{\circ}(p,g) =_{def} \{ hx \in pmf(p,g) \mid \forall d \in DB(\underline{dom}_p), j \in DB(hx) : j \in \underline{dom}_p(d) \Longrightarrow hx(j) \in \underline{dom}_g(d) \}$$

Ein Redexmorphismus hx muss jedes Musterelement j von p so abbilden, dass die Abbildung, hx(j), im Graph g in derselben Domäne d liegt wie j in p.

Definition 2.44: Reduktion mit Domänen
Für das Konzept der Domänen ist die Definition der Funktion reduce *wie folgt anzupassen:*

$$reduce^{\circ}(g,p,hx) =_{def} (g'[\underline{dom} \mapsto dom'], hc)$$
wobei
$$(g',hc) = reduce(g,p,hx)$$
$$dom' = \bigcup d \in DB(\underline{dom}_p) : \{ d \mapsto \underline{dom}_{g'}(d) \cup [\bigcup j \in DB(hc)/DB(hx) \text{ mit } j \in \underline{dom}_p(d) : \{hc(j)\}] \}$$

Die Reduktion wird nur bezüglich des Bestandteils $\underline{dom}$ des erzeugten Graphs g' angepasst. Der neue Wert ist dom' und ergibt sich wie folgt: Für jede Domäne d werden zunächst diejenigen Zuordnungen aus $dom_{g'}(d)$ übernommen, deren Graphelemente noch in g'

vorhanden sind. Hinzu kommen die Graphelemente, die durch den Ersetzungsmorphismus hc in derselben Domäne, d, neu erstellt wurden.

Definition 2.45: Rereduktion mit Domänen
Für das Konzept der Domänen ist die Definition der Funktion rereduce *wie folgt anzupassen:*

$$\text{rereduce}^{\circ}(g,p,hr) =_{def} g'[\underline{dom} \mapsto dom']$$
wobei
$g' = \text{rereduce}(g,p,hr)$
$hx = \text{redexmorph}(hr,p)$
$hc = \text{replacemorph}(hr,p)$
$dom' = \bigcup d \in DB(\underline{dom_p}) : \{ d \mapsto \underline{dom_p}(d) \cup [\bigcup j \in DB(hc)/DB(hx) \text{ mit } j \in \underline{dom_p}(d) : \{hc(j)\}] \}$

Die Anpassung der Rereduktion unterscheidet sich von der Anpassung der Reduktion nur hinsichtlich der Herkunft des Ersetzungsmorphismus, anhand dessen die zu g hinzugefügten Graphelemente einer Domäne zugeordnet werden. Diese Informationen werden aus dem Regelmorphismus hr entnommen.

2.4.2 Transformation mit Graphgrammatiken

Die Verwendung von Domänen ist nicht nur für die Produktionsregeln von Transformationen (vgl. vorheriges Kapitel), sondern auch für solche von Graphgrammatiken interessant: Da die add- und del-Bestandteile nicht auf bestimmte Domänen begrenzt sind, kann jede Domäne als eine eigenständige (Unter-)Graphgrammatik verstanden werden (siehe Abbildung 2.15). Die Graphgrammatik gg1 entspricht dabei dem Beispiel aus Abbildung 2.6 und beschreibt die Menge aller doppelt verketteter Bäume. Die Graphgrammatik gg2 beschreibt die Menge aller „einfachen“ Bäume.

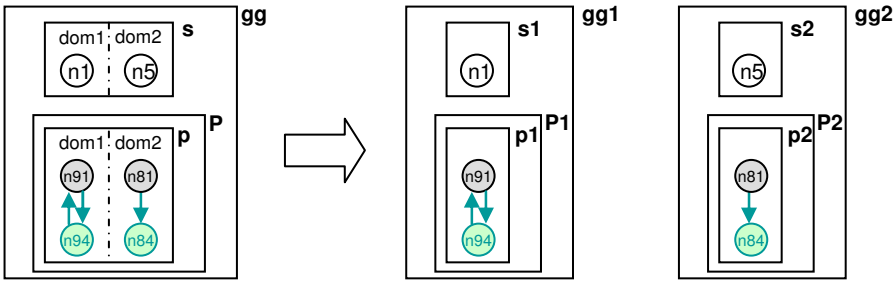


Abbildung 2.15 Domänen einer Graphgrammatik als eigenständige Graphgrammatiken

Der Nutzen der gemeinsamen Graphgrammatik gg besteht darin, dass Wörter beider Sprachen zueinander in Bezug gesetzt werden können. Wie Abbildung 2.16 zeigt, kann beispielsweise aus dem Graph g123 entnommen werden, dass der Graph der ersten Domäne (der zur Sprache von gg1 gehört) dem Graph der zweiten Domäne (der zur Sprache gg2 gehört) „entspricht“. Eine solche gemeinsame Graphgrammatik entspricht in der Literatur dem Begriff der **Paar-Grammatik** (vgl. [39]).

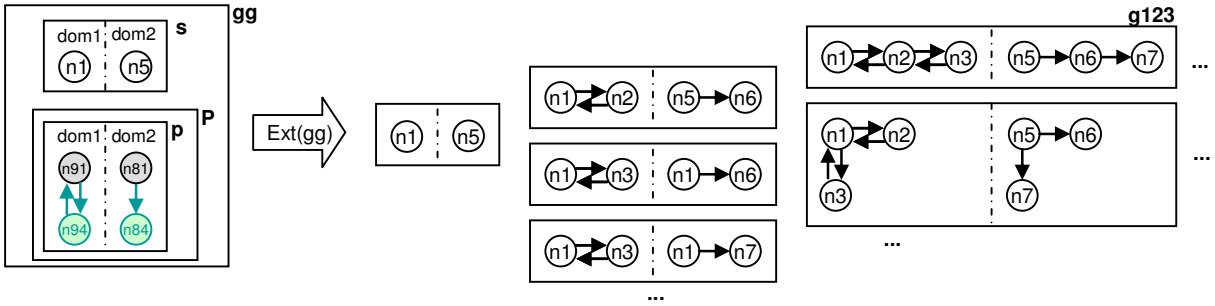


Abbildung 2.16 Extension einer Domänenbasierten Graphgrammatik

Diese Entsprechung kann verwendet werden um Transformationen durchzuführen – hier beispielsweise um einen doppelt verketteten Baum in einen einfachen Baum (oder umgekehrt) umzuwandeln. Einfach formuliert besteht die Transformation darin, den Eingabegraph in der Extension der gemeinsamen Graphgrammatik in der ersten Domäne zu finden, und dann den Graph aus der zweiten Domäne als Ausgabe zurückzuliefern. Ungeachtet der Isomorphien kann es dabei im Allgemeinen mehrere mögliche Ergebnisse geben, so dass eine gemeinsame Graphgrammatik folglich eine Abbildung, und keine Funktion (bzw. Transformation) darstellt. Dies kann jedoch durch eine geeignete Wahl der Produktionsregeln behoben werden. Da es keine Reihenfolge der Domänen gibt – die Rolle der linken oder rechten Domäne also austauschbar ist – beschreiben gemeinsame Graphgrammatiken, bei zwei Domänen, bidirektionale Transformationen.

Ein für die Praxis schwerwiegendes Problem besteht darin, dass die Extension vorab nicht komplett berechnet werden kann, um darin den Eingabegraphen zur Bestimmung der Ausgabe zu suchen. Stattdessen muss eine terminierende Transformation verwendet werden. Abbildung 2.17 zeigt, wie sich diese aus einer gemeinsamen Graphgrammatik ableiten lässt. Der Einfachheit halber seien dabei nur Graphgrammatiken mit nichtlöschenden Produktionsregeln betrachtet. Die Transformation von einer Domäne `a` zu einer Domäne `b` umfasst dann stets alle Produktionsregeln `P` der gemeinsamen Graphgrammatik, wobei in jeder Produktionsregel alle Musterelemente der Domäne `a` dem Suchmuster zugeordnet werden. Je nach Wahl der konkreten Domänen für `a` und `b` entstehen so die beiden Transformationen `T12` und `T21`.

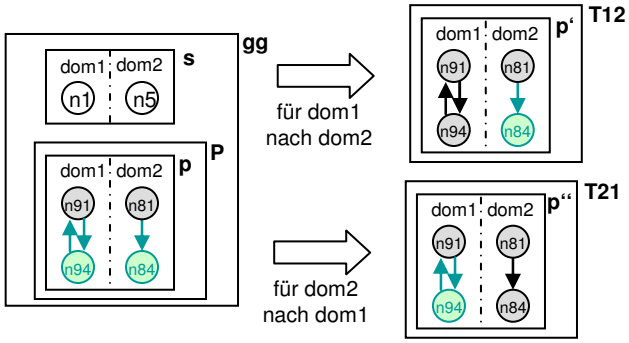


Abbildung 2.17 Ableitung einer Transformation aus einer Graphgrammatik

Die so erhaltene Transformation kann, prinzipiell, auf die Eingabe angewendet werden, um den dazugehörigen Graphen der anderen Sprache zu erhalten, wie in Abbildung 2.18 gezeigt.

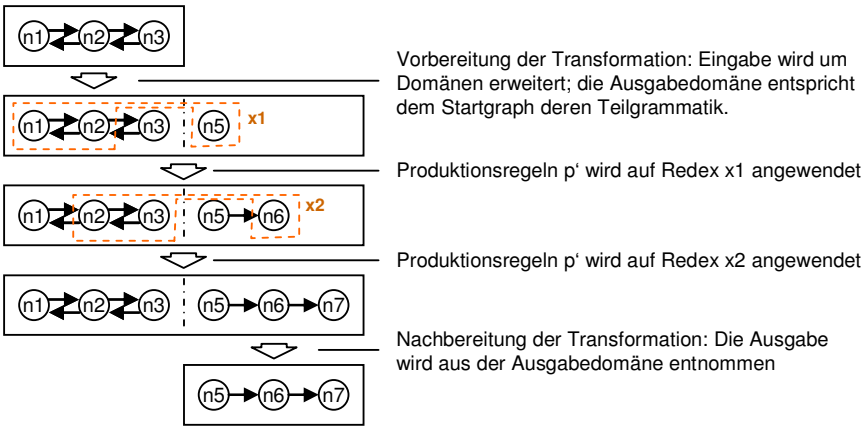


Abbildung 2.18 Ablauf einer Transformation mit einer Graphgrammatik

Bei einer automatisch ausgeführten Transformation ist nicht garantiert, dass der zweite Redex (x2) auch tatsächlich so gewählt wird wie in Abbildung 2.18 gezeigt. Beispielsweise hätte x2 auch das Graphelement n5 (anstelle von n6) umfassen können. Dadurch wäre ein „falscher" Graph entstanden (n6 und n7 hängen an n5), d.h. der nicht gemäß der Extension von gg zur Eingabe zugeordnet ist. Die Ursache liegt darin, dass hier keine Informationen darüber vorliegen, welcher Redex wann und wo verwendet wurde.

Die Lösung besteht darin, diese Informationen explizit bereitzustellen. Da diese Informationen „hinterher", also nach dem die Graphen aus der Graphgrammatik abgeleitet wurden, nicht mehr verfügbar sind, wird die gemeinsame Graphgrammatik um eine weitere Domäne erweitert, in der diese Informationen bei der Erzeugung der Extension auch mit erzeugt werden. Diese weitere Domäne wird in der Literatur als *Correspondence Model* bezeichnet (im Weiteren als Beziehungsdomäne genannt), und die dazugehörige Graphgrammatik als **Tripel-Graph-Grammatik** [40]. Abbildung 2.19 zeigt ein Beispiel einer Tripel-Graph-Grammatik, zusammen mit deren Extension.

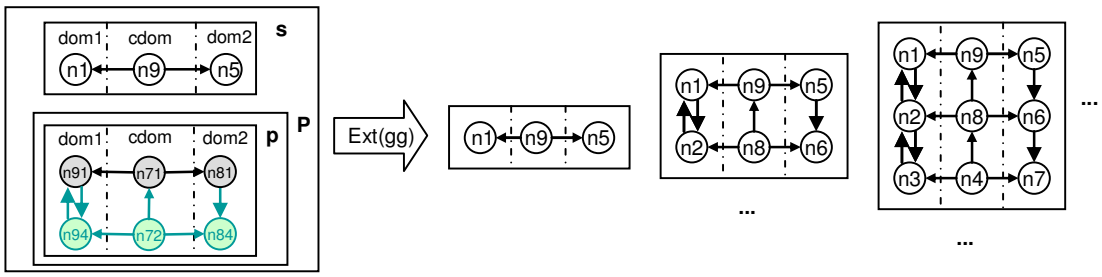


Abbildung 2.19 Tripel-Graph-Grammatik und deren Extension

Abbildung 2.20 zeigt deren Verwendung für eine Transformation. Durch die zusätzlichen Informationen der dritten Domäne kann beispielsweise für den Redex x2 nun nicht mehr die Graphelemente n2 und n3 aus der ersten Domäne zum Graphelement n5 der zweiten Domäne in Bezug gesetzt werden.

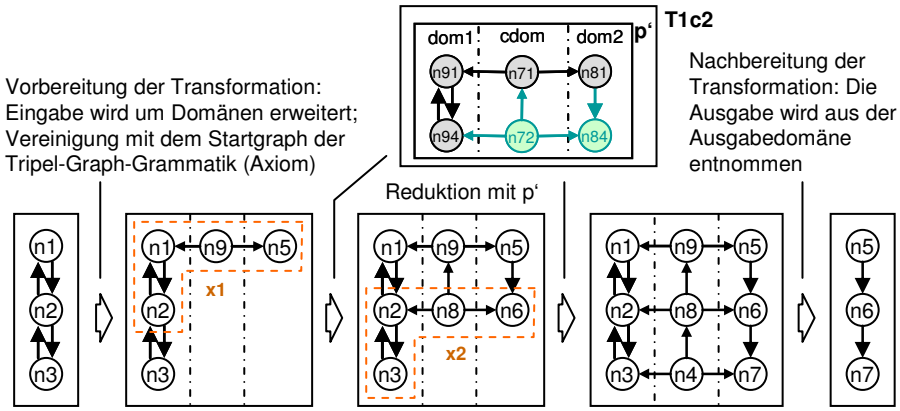


Abbildung 2.20 Anwendungsbeispiel einer Tripel-Graph-Grammatik als Transformation

Ein letztes Problem bei der Nutzung von Graphgrammatiken als Transformationen besteht darin, dass die abgeleitete Transformation nicht terminiert, da einerseits verarbeitete Redexe nicht verschwinden (nichtlöschende Produktionsregeln), und andererseits eine Transformation als ein erschöpfendes Reduzieren aller Redexe definiert ist. So werden bereits für den ersten Redex x1 unendlich viele Elemente im Ausgabegraph erzeugt (vgl. Abbildung 2.21).

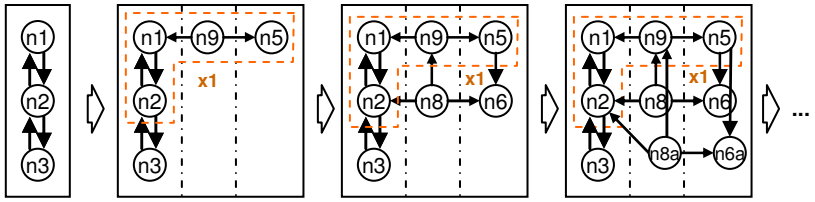


Abbildung 2.21 Details zur Anwendung einer Tripel-Graph-Grammatik als Transformation

Eine in [41][6] beschriebene Lösung dieses Problems besteht darin, diejenigen Knoten der Beziehungsdomäne (Beziehungsknoten) als aktiv zu markieren, zu denen Redexe noch gebildet werden dürfen. Bei jeder Anwendung einer Produktionsregel wird der verwendete Beziehungsknoten deaktiviert, während neu erzeugte Beziehungsknoten zu den aktiven hinzugefügt werden. Wie jedoch pro Anwendung einer Produktionsregel die Terminierung sichergestellt wird, ist nicht näher ausgeführt. Bei einem ähnlichen Algorithmus in [42] wird von einer Ausführung pro Redex (sinngemäß) gesprochen, dies aber auch nicht formal konkretisiert. In dieser Arbeit wird eine einfachere, kompaktere und direkte Möglichkeit dargestellt, um pro Redex und Produktionsregel nur eine Reduktion vorzunehmen. Dazu wird, ähnlich zur inkrementellen Transformation, eine Transformationshistorie mitgeführt, anhand dessen entschieden werden kann, für welche Kombination aus Redex und Produktionsregel eine Reduktion bereits stattgefunden hat. Zudem vereinfacht dieser Ansatz, bezeichnet als redexunikate Transformation, gegenüber [41] und [42] den Algorithmus und gegenüber [41] die Erstellung von Transformationen, da nicht auf eine bestimmte Anordnung der Beziehungsknoten geachtet werden muss.

Definition 2.46: Graphgrammatik mit Domänen
Für das Konzept der Domänen ist die Definition der Menge GG *wie folgt anzupassen:*

$$GG^{\circ} =_{def} \{ (s,P) \in GG \mid [\, \exists\, D \subset DOMAIN : [\, \forall\, g \in \{s\} \cup P : DB(\underline{dom}_g)=D \,] \,] \wedge [\, \forall\, p \in P : \underline{del}_p = \emptyset \,] \,\}$$

Für eine Graphgrammatik $(s,P) \in GG^{\circ}$ *sei die Menge derer Domänen mit* domains((s,P)) *notiert und wie folgt definiert:*

[6] Siehe Kapitel 4.5 der angegebenen Quelle, in dem der Algorithmus für eine nicht-inkrementelle Transformation mittels Tripel-Graph-Grammatiken aufgestellt wird.

$$domains((s,P)) = \bigcup g \in \{s\} \cup P : DB(\underline{dom_g})$$

Weiterhin sei für ein $(s,P) \in GG^{\circ}$ *die Domänengraphgrammatik zu einem* $d \in domains((s,P))$ *mit* $domgg((s,P),d)$ *notiert und wie folgt definiert:*

$$domgg((s,P),d) = (s \restriction_G \underline{dom_g}(d) , \bigcup p \in P : \{ p \restriction_G \underline{dom_g}(d) \})$$

Der Startgraph und alle Produktionsregeln einer Graphgrammatik haben dieselben Domänen und sind alle nicht-löschend. Der Begriff der Domänengrammatik formalisiert die in Abbildung 2.17 Ableitung einer Transformation aus einer Graphgrammatik gezeigte Möglichkeit zur Ableitung einer pro Domäne eigenständigen Graphgrammatik.

Definition 2.47: Ableitung einer Transformation aus einer Graphgrammatik
Gegeben sei eine Graphgrammatik $gg \in GG$ *mit* $gg=(s,P)$*, eine Domäne* $d \in domains(gg)$*. Die* ***Ableitung einer Transformation*** $tr \in TRANS$ *zu* gg *bezüglich* d *wird mit* $gg2tr(gg,d)$ *notiert, und ist wie folgt definiert:*

$$gg2tr((s,P),d) =_{def} tr = \bigcup p \in P : \{ p[\underline{add} \mapsto \underline{add_p}/\underline{dom_p}(d)] \}$$

In jeder Produktionsregel werden alle Musterelemente aus der Domäne d zum Suchmuster zugeordnet.

Definition 2.48: Redexunikate Transformationsanwendung
Gegeben sei eine Transformation $tr \in TRANS$*, und ein Graph* $g0 \in GRAPH$*. Die* ***redexunikate Transformationsanwendung*** *bezüglich* g *und* tr*, notiert als* $trans_1(g,tr)$*, ist wie folgt definiert:*

$$trans_1(g,tr) =_{def} trans_1(g,tr,\emptyset)$$

$trans_1(g,tr,cu)$ = falls $|K|=0$ dann g sonst $trans_1(g'', tr, cu'')$
wobei
$K = [\bigcup p \in tr : \{p\} \times pmf(p,g)] / R$
$R = \bigcup (p,hr) \in cu : \{ (p,redexmorph(hr,p)) \}$
$(p,hx) = rsf(K)$
$(g'',hc) = reduce(g,p,hx)$, einmalig ausgeführt
$cu'' = cu \cup \{(p,hx \cup hc)\}$

Die redexunikate Transformation unterscheidet sich von der normalen Transformation dadurch, dass nach jedem durchgeführten Reduktionsschritt der Verwendete Redexmorphismus in einer Transformationshistorie abgelegt wird. Bei jedem weiteren Reduktionsversuch wird von der Menge der möglichen Redexmorphismen K die Menge R der bereits verwendeten abgezogen. Die redexunikate Transformation ließe sich mit einer inkrementellen Transformation am ehesten nachbilden, wenn jeder Reduktionsschritt der redexunikaten Transformation genau einer Ausführung der inkrementellen Transformation entspräche. Da letztere jedoch bei jeder Ausführung alle Redexe erschöpfend reduziert, ist auch diese Entsprechung nicht zutreffend.

Definition 2.49: Anwendung einer Graphgrammatik als Transformation
Gegeben sei eine Graphgrammatik $gg \in GG$*, zwei Domänen* $a, b \in domains(gg)$ *und ein Graph* $g \in Ext(domgg(gg,a))$*. Die Anwendung von* gg *als Transformation von* a *nach* b *wird mit* $ggtrans(g,gg,a,b)$ *notiert und ist wie folgt definiert:*

$$ggtrans(g,(s,P),a,b) = trans_1(s+g[\underline{dom} \mapsto \{a \mapsto \underline{Id_g}\}] , gg2tr((s,P),a)) \restriction_G \underline{dom_g}(b)$$

Die Anwendung einer Graphgrammatik als Transformation beginnt mit der Vorbereitung des Graphen durch Vereinigung des zu transformierenden Graphen g mit dem Startgraph der Graphgrammatik einerseits, und der Ableitung der Produktionsregeln durch gg2tr andererseits. Nach redexunikater Berechnung des Transformationsergebnisses wird die Nachbereitung durchgeführt, die den Ergebnisgraph aus der Domäne b extrahiert.

Durch den variablen Graph g kann der Startgraph auch als Teil von P geführt werden. Denn gemäß Abbildung 2.5 hat s hier keine besondere Bedeutung mehr. So kann sowohl auf das Integrieren mit g als auch auf die Auszeichnung eines Startgraphs im Allgemeinen verzichtet werden, wenn Graphgrammatiken als Transformationen eingesetzt werden.

Definition 2.50: Tripel-Graph-Grammatik
Die Menge TGG *der* ***Tripel-Graph-Grammatiken*** *ist wie folgt definiert:*

$$
\begin{aligned}
TGG =_{def} \{\, (s,P) \in GG \mid\ & [\ \forall\, g \in \{s\} \cup P: \\
& (DB(\underline{dom}_g) = \{ld,cm,rd\}) \wedge \\
& (\underline{del}_g = \emptyset) \wedge \\
& [\ \forall\, i \in edges(g): \\
& \quad (\ i \in \underline{dom}_g(ld) \Longrightarrow \underline{s}_g(i) \in \underline{dom}_g(ld) \wedge \underline{t}_g(i) \in \underline{dom}_g(ld)\ ) \wedge \\
& \quad (\ i \in \underline{dom}_g(rd) \Longrightarrow \underline{s}_g(i) \in \underline{dom}_g(rd) \wedge \underline{t}_g(i) \in \underline{dom}_g(rd)\ ) \wedge \\
& \quad (\ i \in \underline{dom}_g(cm) \Longrightarrow \underline{s}_g(i) \in \underline{dom}_g(cm)\ )\ ]\ ]\ \}
\end{aligned}
$$

Eine Tripel-Graph-Grammatik ist eine Graphgrammatik bei der der Startgraph und alle Produktionsregeln genau die[7] drei Domänen ld, cm und rd haben. Weiterhin dürfen Kanten die in der Domäne ld liegen, nur Knoten aus ld verbinden, und Kanten in rd nur solche aus rd. Kanten in cm müssen zumindest als Startknoten einen aus cm haben, der Zielknoten kann jedoch in einer anderen Domäne liegen.

2.4.3 Inkrementelle Transformation mit Graphgrammatiken

In diesem Kapitel wird gezeigt, wie Graphgrammatiken für eine inkrementelle Transformation verwendet werden können. Die inkrementelle Transformationsanwendung unterscheidet sich von der rein inkrementellen nur durch die Verwendung einer Transformationshistorie (zur „Wiederherstellung“ von Ids), wodurch sich die Integration beider Themen durch Übertragung dieses Anteils in das Thema der Transformation mit Graphgrammatiken vollziehen lässt. Dieser Ansatz wird im Folgenden als RITG (Redexunikate inkrementelle Transformation mit Graphgrammatiken) bezeichnet.

Der in [17] bzw. [41] beschriebene Ansatz zur Durchführung einer inkrementellen Transformation mit Graphgrammatiken setzt direkt auf Tripel-Graph-Grammatiken auf, und betrachtet damit die inkrementelle Transformation nicht zunächst unabhängig von Graphgrammatiken (also nicht wie dies in dieser Arbeit mit Kapitel 2.3.1 geschehen ist). Konkret äußert sich dies dadurch, dass zur Wiederholung einer Transformation eines zwischenzeitlich ggf. veränderten Graphen die Beziehungsdomäne ausgewertet wird, um zu entscheiden, welche Änderungen am Transformationsergebnis vorzunehmen sind. Im Vergleich zu Kapitel 2.3.1 wird diese Beziehungsdomäne als die Transformationshistorie verwendet. Dabei entspricht jede Kante, die von einem Beziehungsknoten zu einem Knoten einer anderen Domäne führt, einem Eintrag im Regelmorphismus. Dieser Ansatz sei im folgenden als ITTGG (Inkrementelle Transformation mit Tripel-Graph-Grammatiken) bezeichnet. Interessant ist dieser Ansatz deswegen, weil er in bestimmten Situationen mächtiger ist, als RITG. Im Folgenden wird der Mächtigkeitsdefizit identifiziert, und RITG

[7] Die Bennung der Domänen ist austauschbar – zur übersicht werden hier jedoch drei konkrete verwendet.

erweitert um dieses Defizit zu beheben. Darüber hinaus, wie später gezeigt, ist die erweiterte Fassung von RITG für praktische Belange deutlich einfacher als ITTGG anzuwenden.

Zur Verdeutlichung des Mächtigkeitsunterschieds sei das in Abbildung 2.22 vorgestellte Szenario betrachtet: Der Graph g0 wird zunächst mit der Transformation tr inkrementell zum Graph g1 transformiert, anschließend werden in g0 die Graphelemente e2, e4 und n3 durch e6, e5 und n5 ersetzt, wodurch g0' entsteht, und anschließend erneut inkrementell transformiert, wordurch g1' entsteht.

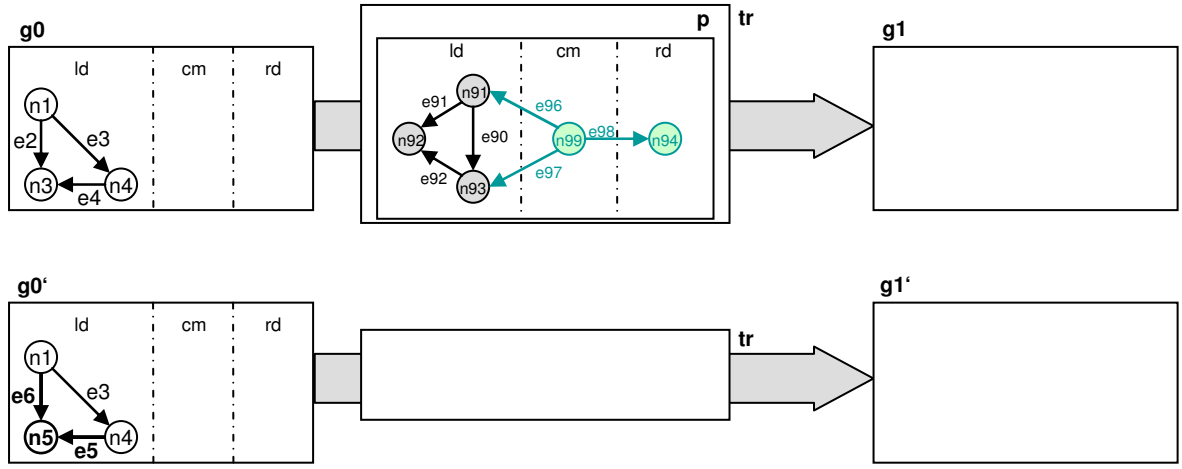


Abbildung 2.22 Szenario zum für inkrementelle Transformation mit Graphgrammatiken

Abbildung 2.23 zeigt das Verhalten von RITG für das in Abbildung 2.22 beschriebene Szenario. Da die Graphelemente e2, e4 und n3 bei der ersten inkrementellen Transformation mit in den Redexmorphismus aufgenommen wurden, führt ein Austausch dieser Elemente bei der zweiten inkrementellen Transformation zum Abbau des in der Transformationshistorie abgelegten Regelmorphismus hr1, und der Erzeugung eines neuen Regelmorphismus hr2. Folglich unterscheidet sich g1' von g1 bezüglich der Graphelemente in den Domänen cm und rd. Zu beachten ist, dass dieses Verhalten völlig im Einklang mit der allgemeinen inkrementellen Transformation (Kapitel 2.3.1) steht; beispielsweise wären die Elemente der Domänen cm und rd zwischen g1' und g1 unverändert geblieben, wenn anstelle des Austauschs von n3 zu n5 (usw.) z.B. ein weiterer Knoten in ld eingefügt worden wäre.

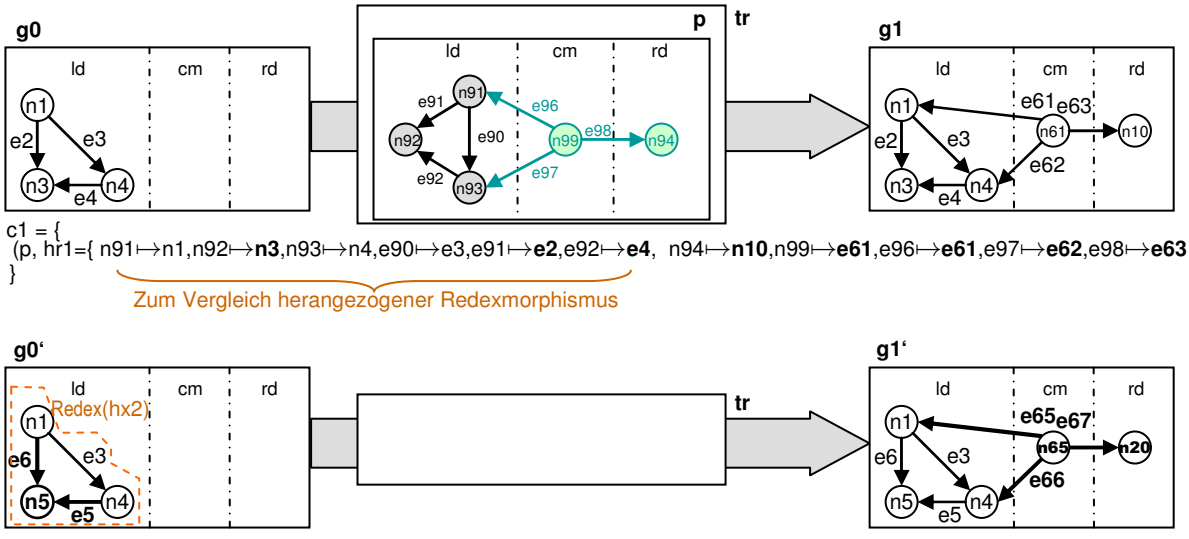


Abbildung 2.23 Inkrementelle Transformation mit Graphgrammatiken mit RITG

Abbildung 2.24 zeigt das Verhalten von ITTGG für das in Abbildung 2.22 beschriebene Szenario. Anders als in RITG wird nicht grundsätzlich erneut transformiert. Stattdessen werden (sinngemäß) die bei der vorherigen Transformation erzeugten Elemente für die Domänen cm und rd direkt in den zu transformierenden Graph g0' eingesetzt, und dann für jeden Beziehungsknoten geprüft, ob er noch den Transformationsregeln von tr entspricht.

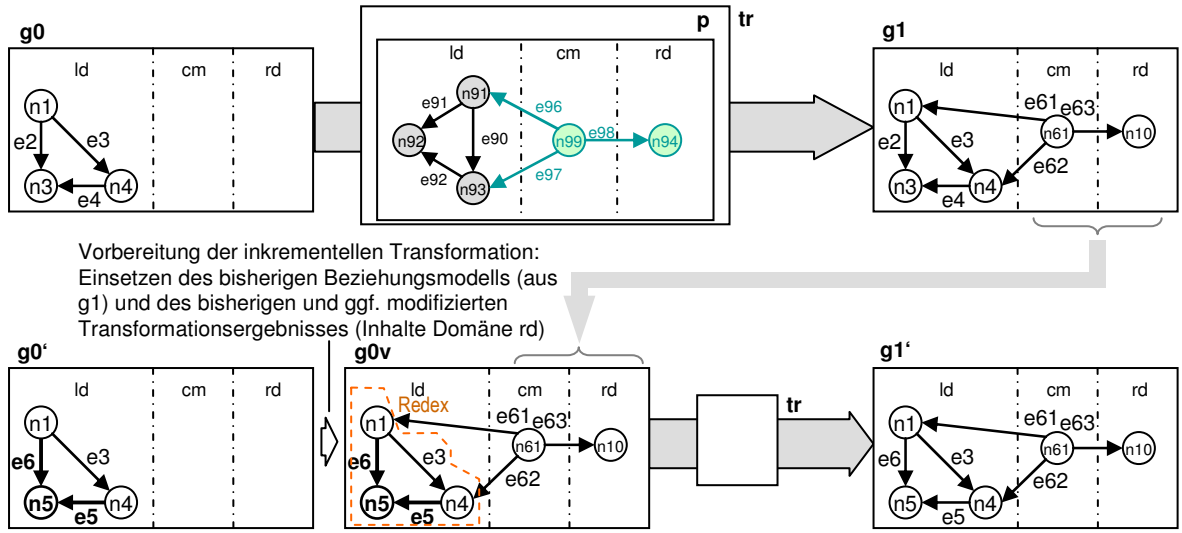


Abbildung 2.24 Inkrementellen Transformation mit Graphgrammatiken mit ITTGG

Da sich hier für den Beziehungsknoten n61 auch nach dem Austausch von n3 zu n5 (usw.) ein Redex finden lässt, wird dieser Beziehungsknoten nicht abgebaut, und damit bleibt auch das Graphelement n10 unverändert. Dieses vom RITG abweichende Verhalten tritt nur dann auf, wenn in einer Produktionsregel nicht alle Suchmusterknoten der Domänen ld und rd von einem Erzeugungsmusterknoten aus cm referenziert werden (was hier für den Musterknoten n92 der Fall ist). Dadurch wird, im Hinblick auf RITG, die Belegung eines solchen Musterknotens nicht mit in die Transformationshistorie aufgenommen. Anders formuliert kann mit ITTGG der Vergleich zwischen neu gefundenen Redexmorphismen und denen die in der Transformationshistorie abgespeichert sind, eingeschränkt werden.

Die Möglichkeit von ITTGG, bei einer inkrementellen Transformation die Redexmorphismen beim Vergleich auf bestimmte Knoten des Suchmusters zu beschränken, kann leicht in RITG integriert werden. Insbesondere kann die Beschränkung auch ohne die Verwendung einer dafür vorgesehenen Domäne eingerichtet werden. Darüber hinaus kann die Auswahl der zu berücksichtigenden Graphelemente nicht nur Knoten, sondern auch Kanten umfassen. Dazu wird die Struktur der Produktionsregeln um den Bestandteil ign (*ign*ore) erweitert, welches angibt, die Belegungen welcher Teile eines Redexmorphismus für die Transformationshistorie zu ignorieren sind. Wie Abbildung 2.25 zeigt, lässt sich diese Information in die Kompaktdarstellung von Produktionsregeln leicht als eine weitere Markierung hinzufügen: Alle Suchmusterknoten deren Belegungen zu ignorieren sind, werden mit blauer Linien- und Textfärbung dargestellt (vgl. Graphelemente e91, e92, n92). Gleichwohl beleibt die Kombinationsmöglichkeit mit Domänen davon unberührt.

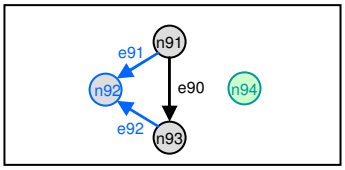


Abbildung 2.25 Notation für den Transformationshistorie zu ignorierende Belegungen

Abbildung 2.26 zeigt das Verhalten der Ignorierung für das in Abbildung 2.22 beschriebene Szenario. Durch die Ignorierung wird ein gefundener und angewendeter Redexmorphismus (z.B. hx1 in der ersten inkrementellen Transformation) nur in entsprechend verkürzter Form im Transformationshistorie abgelegt. Wird ein gefundener Redexmorphismus zur Abklärung dessen Anwendbarkeit mit den bisherigen verglichen, werden ebenfalls nur die nicht zu ignorierenden Anteile geprüft.

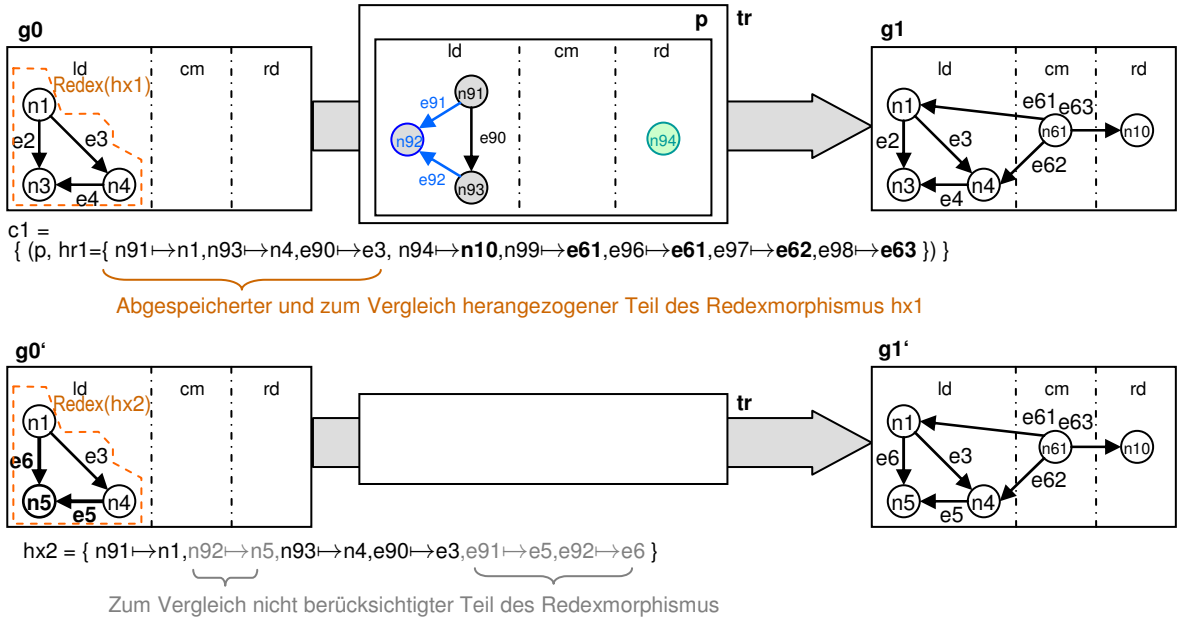


Abbildung 2.26 Verhalten der erweiterten RITG

Abschließend sei anhand zweier Beispiele auf die fachliche Bedeutung der Möglichkeit eingegangen, die in die Transformationshistorie aufzunehmenden Belegungen auswählen zu können.

- Im Vergleich zur Prädikatenlogik ist eine Produktionsregel eine Formel, Musterknoten, deren Belegungen aufzunehmen sind, Quantoren, und solche deren Belegungen nicht aufzunehmen sind, Atome. Die erste Art der Musterknoten spannt damit den Raum gültiger Modelle der Formal auf, während, letztere diesen Raum wieder eingrenzen.
- Im Vergleich zu SQL ist eine Produktionsregel eine SQL-Anfrage, Musterknoten, deren Belegungen aufzunehmen sind, Tabellen in der FROM-Klausel, während solche deren Belegungen nicht aufzunehmen sind, Bedingungen in der WHERE-Klausel. Die erste Art der Musterknoten spannt damit den Raum der betrachteten Zeilen in Form eines Kartesischen Produktes auf, während letztere Zeilen daraus wieder herausstreichen, wodurch das Join-Verhalten nachgebildet wird.

Allgemein stellen in die Transformationshistorie nicht aufzunehmende Suchmusterknoten also Bedingungen dar, die die Zahl der Redexmorphismen reduzieren. Im Folgenden wird daher für die Formalisierung die Festlegung getroffen, dass solche „Bedingungs-Musterknoten“ weder von Erzeugungsmusterknoten referenziert werden dürfen, noch selbst Teils des Löschmusters sein können.

Definition 2.51: Produktionsregel mit Ignorierung

Für das Konzept der inkrementellen Transformation mit Graphgrammatiken ist die Definition der Menge PRODRULE *wie folgt anzupassen:*

$$\text{PRODRULE}^{\circ} =_{def} \{ p \in \text{PRODRULE} \mid \underline{ign_p} \subseteq \underline{Id_p} \wedge$$

$ign_p \cap del_p = \emptyset \wedge$
$ign_p \cap add_p = \emptyset \wedge$
$wellformed< p \restriction_G (l_p/ign_p > \}$

Produktionsregeln werden um den Bestandteil ign erweitert, der, analog zu den Bestandteilen add und del, Elemente des Suchmusters als solche markieren kann, deren Belegungen in die Transformationshistorie nicht aufzunehmen sind. Beim Vergleich der Redexmorphismen im Rahmen einer inkrementellen Tranformation werdn diese ignoriert. Dabei gilt, dass unter Weglassung der zu ignorierenden Graphelemente, die Produktionsregel wohlgeformt bleiben muss. Diese Bedingung setzt die Erkenntnis um, dass die zu ignorierenden Graphelemente lediglich die anwendbaren Redexmorphismen einschränken.

Definition 2.52: Patternmatchingfunktion mit Ignorierung
Für das Konzept der inkrementellen Transformation mit Graphgrammatiken ist die Definition der Patternmatchingfunktion wie folgt anzupassen:

$$pmf^{\circ}(p,g) =_{spec} \bigcup hx \in pmf(p,g) : \{ hx \restriction (ld_p/ign_p) \}$$

Wird das Konzept der inkrementellen Transformation in Kombination mit anderen Konzepten verwendet, so ist diese Anpassungsdefinition als letzte durchzuführen.

Durch die Ignorierung werden die Redexmorphismen auf den Teil beschränkt, der in die Transformationshistorie aufzunehmen ist. Diese ermitteln sich aus der Differenz von ld_p und ign_p. Die Bedingung, diese Anpassungsdefinition bei Einsatz weiterer Konzepte als letzte auszuführen, sichert, dass Redexmorphismen von anderen Konzepten nicht mehr ausgeschlossen werden können, da sie nach der hier vorgenommenen Beschränkung eine andere Entscheidungsgrundlage darbieten.

2.4.4 Negationsbereich

Eine weitere Möglichkeit (neben Domänen) zur Einschränkung der Anwendbarkeit von Produktionsregeln sind Negationsbereiche. Ein Negationsbereich ist eine Fläche (rot-grau-schraffiert, vgl. Abbildung 2.27) die eine beliebige Anzahl von Graphelementen (hier n99 und e99) einer Produktionsregel (hier p) umfassen kann. Wenn *alle* so umfassten Graphelemente als Teil eines Redexmorphismus belegt werden können, dann ist die Produktionsregel für diesen Redexmorphismus nicht anwendbar.

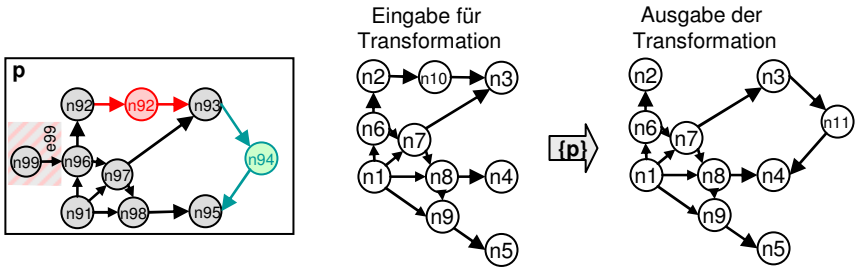


Abbildung 2.27 Beispiel des Einsatzes Negationsbereiches

Zur Formalisierung wird für Produktionsregeln der Bestandteil neg eingeführt, der eine Menge von Negationsbereichen enthält. Eine Produktionsregel mit mehreren Negationsbereichen ist genau dann für einen Redexmorphismus nicht anwendbar, wenn mindestens ein Negationsbereich vollständig belegt werden kann.

Im Unterschied zu Ansätzen aus der Literatur (wie beispielsweise in [43]), sind in dieser Arbeit Negationsbereiche hierarchisch. Ein Negationsbereich kann also wiederum mehrere Negationsbereiche enthalten (vgl. Abbildung 2.28).

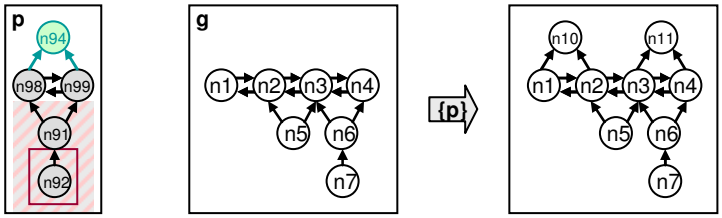


Abbildung 2.28 Notation und Wirkung hierarchischer Negationsbereiche

Definition 2.53: Negationsbereich
Die Menge NEG *aller* ***Negationsbereiche*** *ist die kleinste Menge die folgende Spezifikation erfüllt:*

$$n \subseteq GE \Longrightarrow_{spec} n \in NEG$$
$$n \subseteq GE \wedge subs \subseteq NEG \Longrightarrow_{spec} n \cup subs \in NEG$$

Für ein Negationsbereich $n \in NEG$ *bestimmt sich die Menge aller darin enthaltener Teilbereiche, notiert mit* negpot(n)*, wie folgt:*

$$negpot(n) =_{def} n \cup [\bigcup_{sub \in n \cap NEG} negpot(sub)]$$

Für ein Negationsbereich $n \in NEG$ *wird die Menge aller rekursiv enthaltener Graphelemente als dessen Hülle bezeichet, mit* neghull(n) *notiert, und bestimmt sich wie folgt:*

$$neghull(n) =_{def} (n \cap GE) \cup [\bigcup_{sub \in n \cap NEG} neghull(sub)]$$

Ein Negationsbereich ist eine Menge, die Graphelemente und Negationsbereiche enthält.

Definition 2.54: Produktionsregel mit Negationsbereichen
Für das Konzept der Negationsbereiche ist die Definition der Menge PRODRULE *wie folgt anzupassen:*

$$PRODRULE^{\circ} =_{def} \{ p \in PRODRULE \mid$$
$$\underline{neg}_p \subseteq NEG \wedge$$
$$neghull(\underline{neg}_p) \cap \underline{del}_p = \emptyset \wedge$$
$$neghull(\underline{neg}_p) \cap \underline{add}_p = \emptyset \wedge$$
$$[\forall n \in negpot(\underline{neg}_p) : wellformed< p \restriction_G (\underline{ld}_p/neghull(n)) >] \}$$

Eine Produktionsregel wird um den Bestandteil neg erweitert, der eine Menge von Negationsbereichen enthält[8]. Die Hüllen aller Negationsbereiche sind zu denen als zu löschen oder zu hinzuzufügen disjunkt.

Definition 2.55: Patternmatchingfunktion mit Negationsbereichen
Für das Konzept der Negationsbereiche ist die Definition der Patternmatchingfunktion wie folgt anzupassen:

[8] Eine Menge von Negationsbereichen ist selbst wieder ein Negationsbereich. Formal ist $\underline{neg}_p$ also auch ein Negationsbereich. Dies wird zwar ausgenutzt, um die Bedingungen an mit Negationsbereichen erweiterte Produktionsregeln kürzer formulieren zu können. Durch die Anpassung der Patternmatchingfunktion (siehe nachfolgende Definiton) wertet $\underline{neg}_p$ selbst aber nicht als Negationsbereich – sondern nur dessen Teilmengen.

$$pmf°(p,g) =_{spec} pmf(p|_G\, v,g) \,/\, [\bigcup n \in \underline{neg_p} : pmf°(\,(p|_G\, s)[\underline{neg} \mapsto n \cap NEG]\,,\, g\,)]$$

wobei

$v = \underline{Id_p} \,/\, neghull(\underline{neg_p})$

$s = v \cup neghull(n)$

Zunächst wird mit v der Teil des Kontextmusters von p bestimmt, der nicht zu negieren ist; d.h. der auf jeden Fall von anwendbaren Redexmorphismen zu belegende Teil. Über die ursprüngliche Definition der Patternmatchingfunktion wird dann die Menge aller dazu passenden Redexmorphisme ermittelt (durch $pmf(p|_G v,g)$). Davon werden diejenigen abgezogen, die neben v zusätzlich auch noch einen Negationsbereich n vollständig belegen. Letzteres wird rekursiv berechnet, wobei pro Rekursionsschritt eine Negationsebene aufgelöst wird. Abbildung 2.29 verdeutlicht die Berechnung anhand des Beispiels aus Abbildung 2.28.

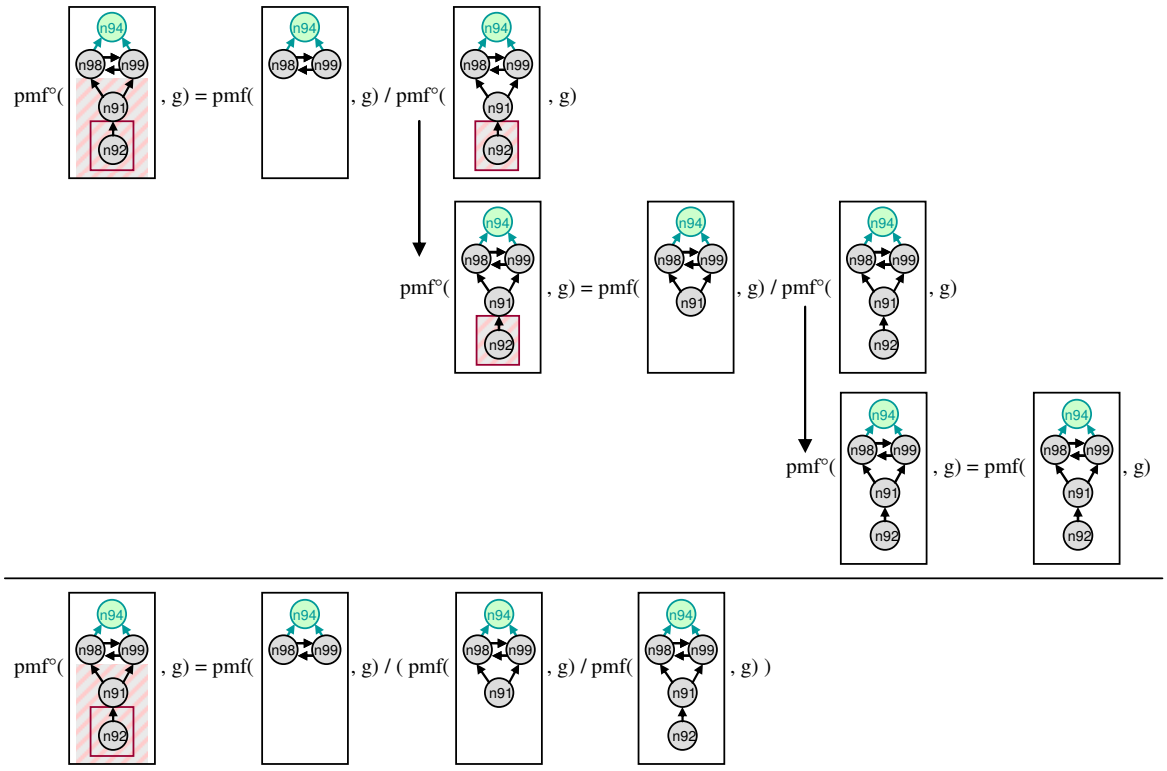


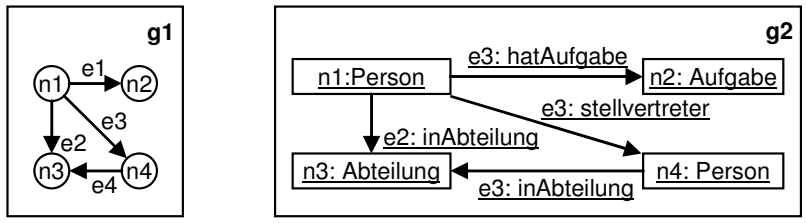
Abbildung 2.29 Anpassung der Patternmatchingfunktion für Negationsbereiche

2.4.5 Typ

Knoten oder Kanten mit gleicher fachlicher Bedeutung sollten in der Praxis explizit als solche gekennzeichnet und erkannt werden können. Dazu werden Typen verwendet, wobei in einem Graph jedem Graphelement genau ein Typ zugeordnet wird. Diese Kennzeichnungen sind auch im Hinblick auf Produktionsregeln von Vorteil: Anstelle fachlich verschiedene Knoten mit unterschiedlichen, dranhängenden Graphstrukturen für Produktionsregeln unterscheidbar zu machen, genügt eine einzige Kennzeichnung mit einem Typ (siehe auch [44,45]).

Wie Abbildung 2.30 zeigt, wird zur Darstellung von Typen die Notation von UML übernommen: der Typ eines Knotens wird als Teil desselben unmittelbar hinter der Id des Knotens, und durch einen Doppelpunkt getrennt, platziert. Der Zusammenhang aus Knotennamen und Knotentyp wird zusätzlich durchgängig unterstrichen. Analog wird auch mit dem Typ einer Kante verfahren. Ferner werden getypte Knoten als Rechtecke dargestellt.

Die Angabe der Id und/oder des Typs kann weggelassen werden, sofern diese in der Darstellung nicht relevant sind.



ungetypter Graph

getypter Graph,
verwendete Typen:
Person, Aufgabe, Abteilung,
hatAufgabe, inAbteilung, stellvertreter

Abbildung 2.30 Darstellung ungetypter und getypter Graphen im Vergleich

Bei (inkrementellen) Transformationen wirken Typen als eine Beschränkung der gültigen Redexmorphismen. Da Typen nicht nur die Struktur der Produktionsregeln, sondern bereits die der Graphen durch einen weiteren Bestandteil erweitert, wird bei der Formalisierung bereits bei der Anpassung des Graphhomomorphismus angesetzt. Dies wirkt sich implizit auch auf die gültigen Redexmorphismen aus (siehe Abbildung 2.12).

Definition 2.56: Graphelementtyp
Die Menge GETYPE *ist die Menge aller* ***Graphelementtypen****, kurz Typen.*

Bei Graphelementtypen wird nicht zwischen Typen für Knoten und Kanten unterschieden. Im Rahmen dieser Arbeit werden gemäß der Spezifikation eine Reihe reservierter Typen verwendet, deren Bedeutung an gegebener Stelle erläutert wird. Um Namenskollisionen mit Bestandteilen zu vermeiden, werden für Typen deutsche Bezeichnungen verwendet.

Definition 2.57: Getypter Graph
Für das Konzept der Typen ist die Definition der Menge GRAPH *wie folgt anzupassen:*

$$GRAPH^{\circ} =_{def} \{ g \in GRAPH \mid \underline{y}_g : \underline{Id}_g \rightarrow GETYPE \}$$

Ein Graph $g \in GRAPH$ enthält $\underline{y}$ als weiteren Bestandteil mit dem jedem Graphelement ein Typ zugewiesen werden kann.

Definition 2.58: Graphhomomorphismus für getypte Graphen
Für das Konzept der Typen ist die Definition des Graphhomomorphismus[9] *wie folgt anzupassen:*

$$homomorph^{\circ}<h,g1,g2> =_{def} homomorph<h,g1,g2> \wedge [\ \forall\ i \in \underline{Id}_{g1} \cap DB(\underline{y}_{g1}) : \underline{y}_{g1}(i) = \underline{y}_{g2}(h(i))\ ]$$

Zu beachten ist, dass ein Graphmorphismus nicht den Typ verändert, so dass ein Graphelement i in $g1$ und dessen Abbildung $h(i)$ in $g2$ tatsächlich *denselben* Typ haben müssen. Eine Ausnahme besteht lediglich darin, wenn ein Graphelement i in $g1$ keinen Typ zugeordnet bekommen hat – dann passt ein beliebiger Typ für $h(i)$.

[9] Durch die Anpassung des Graphhomomorphismus werden implizit auch die Definitionen für die Patternmatchingfunktion und die Erzeugungsfunktion angepasst. Die Anpassung erfolgt hier bereits auf Ebene der Graphen (und nicht auf Ebene der Produktionsregeln), da Typen als Bestandteil von Graphen eingeführt sind.

2.4.6 Primitivwert

Nicht alle Informationen lassen sich effizient durch die Struktur von Knoten und Kanten bzw. Typen darstellen. Wie Abbildung 2.31 zeigt, gilt dies insbesondere für Zeichenketten, die im Weiteren auch als Primitivwerte bezeichnet werden. Im Graph g1 wird die Zeichenkette „EINTEXT“ durch eine Kette von Knoten dargestellt, wobei jeder Knoten durch dessen Typ genau ein Zeichen repräsentiert. Die Menge der Typen ist dabei fest und entspricht z.B. der Menge der ASCII-Zeichen [46]. Im Graph g2 wird eine Zeichenkette dagegen vollständig durch einen Typ dargestellt. Da es unendlich viele Zeichenketten gibt, müssen auch unendlich viele Typen vorgesehen werden. Das wiederum ist im Rahmen einer Sprachdefinition durch eine Graphgrammatik nicht darstellbar: Mit Produktionsregeln können nur neue Graphelement-Geflechte (mit vorgegebenen Typen), aber keine neuen Typen erzeugt werden. Außerdem besteht immer noch das Problem, in einer Sprachdefinition nicht vorgeben zu können, dass nur bestimmte Knoten eine Zeichenkette sein sollen.

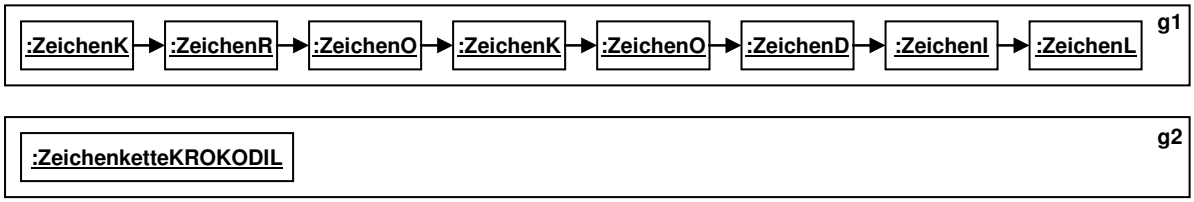


Abbildung 2.31 Zeichenketten als Graphen unter Verwendung von Typ pro Zeichen

Eine dritte, in dieser Arbeit verfolgte Möglichkeit besteht in der Nutzung von Ids als Informationsträger (Abbildung 2.32). Entsprechend muss die Menge GE alle Zeichenketten umfassen.

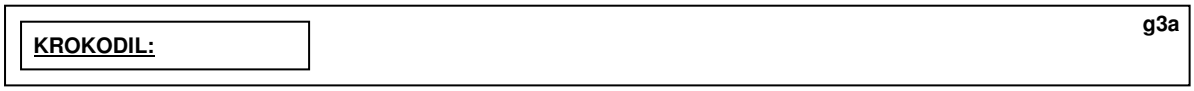


Abbildung 2.32 Zeichenketten als Ids

Durch die Verwendung des reservierten Typs Zeichenkette (Abbildung 2.33) kann die Extension einer Graphgrammatik bzw. das Verhalten einer Transformation so angepasst werden, dass für ein Graphelement jenes Typs nur bestimmte Ids als Belegung in Frage kommen. Mit dieser Technik können weitere Arten von Primitivwerten wie beispielsweise Zahlen oder Wahrheitswerte abgeleitet werden.

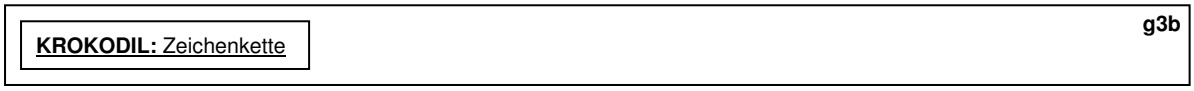


Abbildung 2.33 Zeichenketten als Ids mit reserviertem Typ

Der reservierte Typ Zeichenkette kann dabei in Kombination mit anderen Typen verwendet werden. Die Nutzung dieses Typs ist jedoch nicht zwingend wie die Gegenüberstellung in Abbildung 2.34 zeigt.

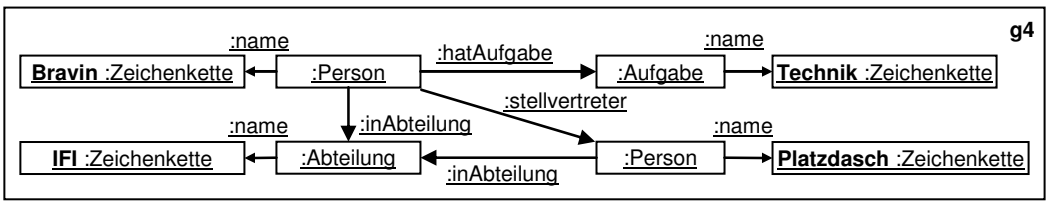


Abbildung 2.34 Zeichenketten in gemischter Verwendung mit anderen Typen

Die Formalisierung basiert gemäß der Nutzung von Typen auf dem im vorangegangen Kapitel vorgestellten Konzept der Typen und nutzt als Bezugspunkt das entsprechend um Typen erweiterte Grundkonzept.

Definition 2.59: Zeichen, Zeichenkette
Sei CHAR *die Menge aller* ***Zeichen****. Die Menge aller* ***Zeichenketten*** *wird als* STRING *bezeichnet und ist die kleinste Menge, die folgende Spezifikation erfüllt:*

$STRING \subseteq_{spec} GE$
$CHAR \subseteq_{spec} STRING$
$c \in CHAR \wedge s \in STRING \Longrightarrow_{spec} (c,s) \in STRING$

Als Notationen sei folgendes vereinbart:

$(c1,c2) \Leftrightarrow c1 \cdot c2 \Leftrightarrow c1c2$
$'eintext' \Longrightarrow eintext \in STRING$

Die erste Spezifikationsbedingung besagt, dass jede Zeichenkette auch in GE enthalten sein muss, um sie als Id eines Knotens nutzen zu können. GE muss folglich entsprechend ausgelegt werden. Die zweite Bedingung besagt, dass jedes Zeichen aus CHAR eine Zeichenkette ist, während die dritte Bedingung besagt, dass jedes Tupel aus einem Zeichen c und einer Zeichenkette s wiederum als eine Zeichenkette angesehen wird. Zur Notation kann anstelle des Tupels entweder infix der sogenannte und assoziative Konkatenationsoperator (vgl. [33]) verwendet, oder eine Verknüpfung der Zeichen ganz weggelassen werden. Ein in Hochkommas gestelltes s bedeutet, dass s ein Element von STRING ist.

Definition 2.60: Patternmatchingfunktion mit Primitivwerten
Für das Konzept der Primitivwerte ist die Spezifikation der Patternmatchingfunktion pmf *wie folgt anzupassen:*

$pmf°(p,g) =_{spec} \{ hx \in pmf(p,g) \mid \forall j \in \underline{Id}_p \text{ mit } \underline{y}_p(j)=Zeichenkette : hx(j) \in STRING \}$

Die Spezifikation der Funktion pmf wird um die Bedingung erweitert, dass jeder gefundene Redexmorphismus hx ein Graphelement j mit Typ Zeichenkette auf eine Zeichenkette abbilden muss. Die Abbildung von j, hx(j), muss also ein Element von STRING sein.

2.4.7 Operation

Mit Produktionsregeln lassen sich Operationen auf Graphen beschreiben. Durch Darstellung der Primitivwerte als Ids fehlt es jedoch an Operationen über selbigen: Produktionsregeln können bisher nur neue Geflechte von Graphelementen erzeugen, nicht jedoch die Belegung einer Id in Abhängigkeit einer Operation (z.B. Addition) bestehender Ids beschreiben. Abbildung 2.35 zeigt an einem Beispiel, wie dieser Mangel behoben wird: eine Produktionsregel kann im Suchmuster Variablen (in der Darstellung mit dem Symbol $ beginnend) verwenden um auf die Ids existierender Graphelemente zuzugreifen, und diese im Erzeugungsmuster in einem Term zu referenzieren. Ein Term kann dabei eine Operation (hier: concat) verwenden um die konkreten Ids zu verarbeiten und eine neue zu berechnen.

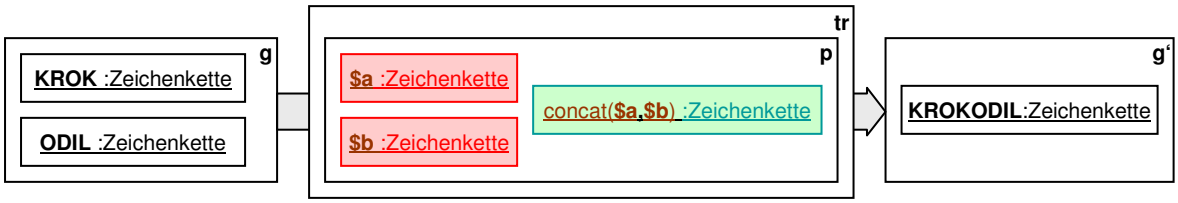


Abbildung 2.35 Beispiel einer Operation auf Primitivwerten

Da das Verhalten einer Operation beliebig festgelegt werden kann, könnte bei der Erzeugung eine Id vergeben werden, zu dem es im Graph bereits einen Knoten gibt. Beispielsweise geschieht dies bereits bei einer einfachen Übertragung von Graphelementen aus einer Domäne in eine andere, wie in Abbildung 2.36 gezeigt.

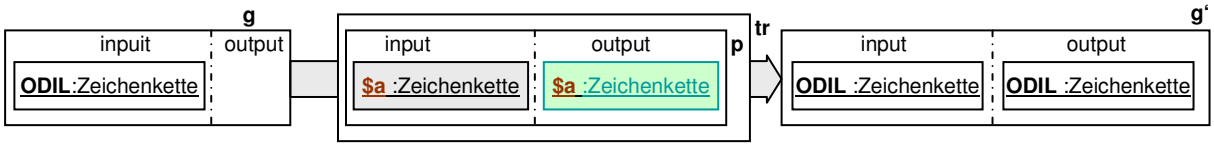


Abbildung 2.36 Kopieren von Graphelementen zwischen Domänen

Um eine Kollision mit erzeugten Ids zu vermeiden, wird eine Transformation in zwei Stufen durchgeführt: In der ersten Stufe werden neue Ids vergeben, zu denen über eine Abbildung die tatsächlich zu setzende Id zugeordnet wird (siehe Abbildung 2.37, idmap). In der zweiten Stufe, nach Durchführung aller Reduktionen, kann die Abbildung aufgelöst werden (Abbildung 2.37, Funktion resolveid), wobei zuvor etwaige Umformungen des Graphen vorgenommen werden um Kollisionen zwischen Ids zu vermeiden. Konkret wird der Graph g' auf die Domäne output beschränkt.

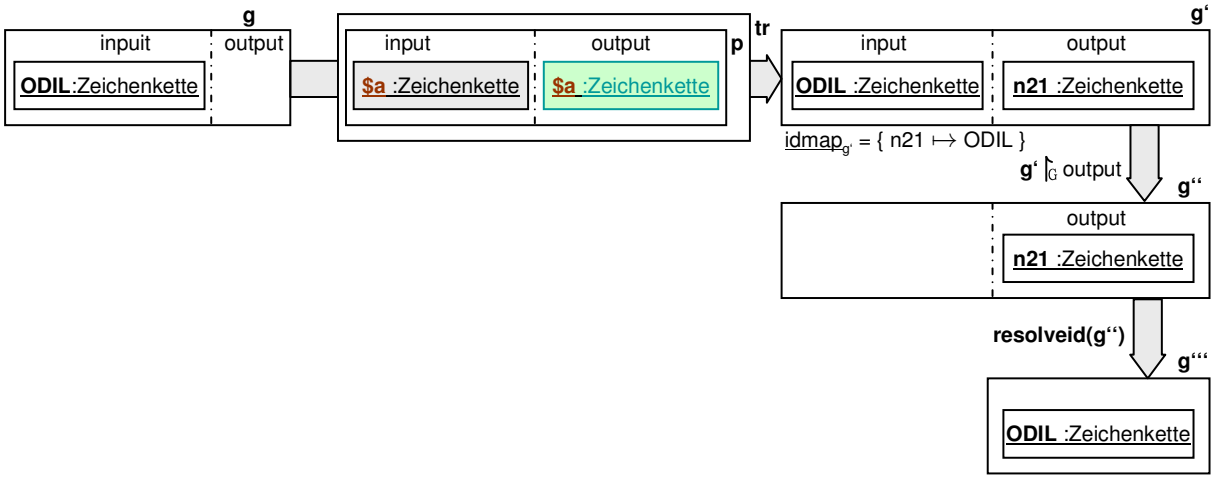


Abbildung 2.37 Verwendung einer Id-Abbildung im Rahmen von Operationen auf Ids

Die Terme werden durch den neuen Bestandteil term für Produktionsregeln abgebildet, der jedem Graphelement einer Produktionsregel einen Term zuweisen *kann*. Wie Abbildung 2.38 zeigt, wird zur Darstellung der Term direkt als Id des entsprechenden Knotens gesetzt.

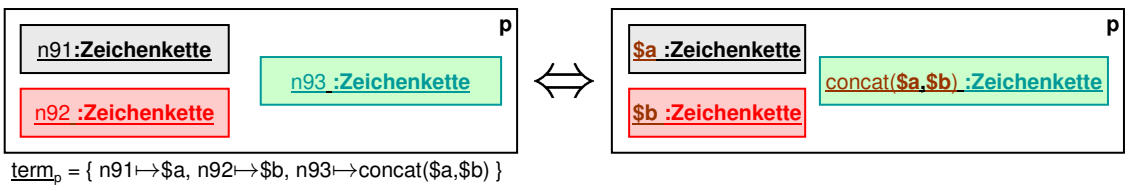


Abbildung 2.38 Notation für Terme

Definition 2.61: Termvariable
Die Menge TVAR *sei die Menge aller* ***Termvariablen****.*

Definition 2.62: Operationssymbol
Die Menge OPS *bezeichnet die Menge aller* ***Operationssymbole****. Im Rahmen dieser Arbeit erfüllt die Menge* OPS *die folgende Spezifikation:*

$$\{ \underline{concat} \} \subseteq_{spec} OPS$$

Definition 2.63: Term, Grundterm
Die Menge TERM *aller* ***Terme*** *ist die kleinste Menge die folgende Spezifikation erfüllt:*

$$GE \subseteq_{spec} TERM$$
$$TVAR \subseteq_{spec} TERM$$
$$t1,\ldots,tn \in TERM \wedge op \in OPS \Longrightarrow_{spec} (op,t1,\ldots,tn) \in TERM$$

Als Notation sei folgendes vereinbart:

$$(op,t1,\ldots,tn) \Leftrightarrow op(t1,\ldots,tn)$$

Die Variablen eines Terms $t \in TERM$ *werden mit* Vars(t) *notiert und sind wie folgt definiert:*

$$Vars(t) =_{def}$$
falls $t \in GE$ dann $\emptyset$
falls $t \in TVAR$ dann $\{ t \}$
falls $t=(op,t1,\ldots,tn)$ dann $Vars(t1) \cup \ldots \cup Vars(tn)$

Die Menge GTERM *der* ***Grundterme*** *ist die Menge aller Terme ohne Variablen:*

$$GTERM =_{def} \{ t \in TERM \mid Vars(t) = \emptyset \}$$

Jede Id eines Graphelements und jedes Variablensymbol ist auch ein Term. Jede Menge aus Termen, auf die ein Operationssymbol $op \in OPS$ angewendet wird, bildet ebenfalls einen Term. Zur Einfachheit wird hier auf die Betrachtung und Einhaltung der Stelligkeit eines Operationssymbols verzichtet. Als Notation wird vereinbart, dass die Operation auch vor deren Argumente geschrieben werden kann – so *als ob* sie angewendet werden würde. Die tatsächliche Anwendung erfolgt ausschließlich innerhalb der Termauswertungsfunktion (siehe entsprechende noch folgende Definition).

Definition 2.64: Produktionsregel mit Operationen
Für das Konzept der Operationen ist die Definition der Menge PRODRULE *wie folgt anzupassen:*

$$PRODRULE^{\circ} =_{def} \{ p \in PRODRULE \mid$$
$$\underline{term}_p \in \{ f : \underline{Id}_p \rightarrow TERM \} \wedge$$
$$[\forall j \in \underline{Id}_{searchpattern(p)} \cap DB(\underline{term}_p) : \underline{term}_p(j) \in TVAR] \ \}$$

Produktionsregeln werden um den Bestandteil term erweitert, der jedem Musterelement einen Term zuweisen kann. Dabei darf einem Suchmusterelement nur eine Variable als Term zugewiesen werden. Zusätzlich gilt die Bedingung, dass die in den Termen des Ersetzungsmusters genutzten Variablen als Terme im Suchmuster vorkommen müssen. Diese

Bedingung wird jedoch erst in der Anpassung der Patternmatchingfunktion formal abgelegt, da anderernfalls die Funktionen searchpattern und replacepattern nicht unter PRODRULE abgeschlossen wären.

Definition 2.65: Termauswertungsfunktion
*Die **Termauswertungsfunktion*** tef *ist eine Funktion, die zu einem Grundterm* t ∈ GTERM *als Eingabe, ein Graphelement aus* GE *als Ausgabe liefert. Im Rahmen dieser Arbeit ist sie wie folgt spezifiziert:*

$\forall$ t∈GTERM : tef(t) $=_{spec}$ falls t = concat(t1,…,tn) dann concat(tef(t1), … tef(tn)) sonst t
wobei
s1 ∈ CHAR $\wedge$ s2 ∈ STRING $\Longrightarrow$ (concat(s1,s2) = (s1,s2))
c1 ∈ CHAR $\wedge$ s1,s2 ∈ STRING $\Longrightarrow$ (concat((c1,s1),s2) = (c1,concat(s1,s2)))

Die Termauswertungsfunktion beschreibt wie Grundterme zu verarbeiten sind. Die erste Spezifikationsbedingung fordert, dass ein Graphelement als Term auf sich selbst abgebildet wird – ohne Operation also keine Veränderung der Eingabe stattfindet. Die zweite Spezifikationsbedingung beschreibt das Verhalten der Operation concat mit Hilfe der Funktion concat (vgl. Semantik des Konkatenationsoperators in [33]). Da die Termauswertungsfunktion als Spezifikation und nicht als Definition aufgestellt ist, können weitere Operationen transparent hinzugefügt werden.

Definition 2.66: Patternmatchingfunktion mit Operationen
Für das Konzept der Operationen ist die Spezifikation der Patternmatchingfunktion pmf *wie folgt anzupassen:*

pmf°(p,g) $=_{spec}$ { hx ∈ pmf(p,g) |
[$\bigcup$ j∈$\underline{ld}_{replacepattern(p)}$∩DB($\underline{term}_p$) : { Vars($\underline{term}_p$(t))] ⊆
[$\bigcup$ j∈$\underline{ld}_{searchpattern(p)}$∩DB($\underline{term}_p$) : { $\underline{term}_p$(j) }] $\wedge$
|[$\bigcup$ j∈$\underline{ld}_{searchpattern(p)}$∩DB($\underline{term}_p$) : { $\underline{term}_p$(j) }]| =
|[$\bigcup$ j∈$\underline{ld}_{searchpattern(p)}$∩DB($\underline{term}_p$) : { ($\underline{term}_p$(j),hx(j)) }]| }

Wird eine Variable mehreren Suchmusterknoten als Term zugewiesen, so muss ein Redexmorphismus hx alle diese Suchmusterknoten mit *demselben* Graphelement aus g belegen. Formal ausgedrückt muss die Anzahl der Variablen im Suchmuster gleich der Anzahl der Paare aus Variable und deren Belegung sein.

Definition 2.67: Variablenauflösung
Gegeben sei ein Term t ∈ TERM*, ein Graph* g ∈ GRAPH*, eine Produktionsregel* p ∈ PRODRULE *und ein Redexmorphismus* hx ∈ REDEXMORPH*. Die **Variablenauflösung** überführt einen Term* t *in einen Grundterm, indem sämtliche Variablen in* t *anhand von* p *und* hx *durch Graphelemente aus* g *ersetzt werden, und ist wie folgt definiert:*

resolvevar(t,g,p,hx) $=_{def}$
falls t ∈ GE dann t
falls t ∈ TVAR dann any($\bigcup$ j∈$\underline{ld}_{searchpattern(p)}$ mit $\underline{term}_p$(j)=t : { hx(j) })
falls t = (op,t1,…,tn) dann (op, resolvevar(t1,g,p,hx), … , resolvevar(tn,g,p,hx))

Sofern ein Term t eine Id ist, wird t nicht verändert. Wenn t eine Variable ist, wird diese durch dasjenige Graphelement aus g ersetzt, auf das der Redexmorphismus hx den mit t versehenen Suchmusterknoten j abbildet. Wenn t sich aus einem Operationssymbol und einer Menge von Teiltermen t1 … tn zusammensetzt, werden zunächst die Teilterme mit einer

Variablenauflösung behandelt, und anschließend die ursprüngliche Anwendung der Operation op wieder zusammengesetzt.

Definition 2.68: Reduktion mit Operationen
Für das Konzept der Operationen ist die Definition der Funktion reduce *wie folgt anzupassen:*

$$\text{reduce}^{\circ}(g,p,hx) =_{def} (g'',hc)$$
wobei
$(g',k) = \text{reduce}(g,p,hx)$, einmalig ausgeführt
$g'' = g'[\underline{idmap} \mapsto \underline{idmap}_{g'} \cup E]$
$E = \bigcup_{j \in (DB(hc)/DB(hx)) \cap DB(\underline{term}_p)} : \{ hc(j) \mapsto tef(resolvevar(\underline{term}_p(j),g',p,hx)) \}$

Die Reduktion wird dahingehend angepasst, das der erzeugte Graph g' den zusätzlichen Bestandteil idmap erhält und so zu g'' wird. Sofern bei einer Transformation mit mehreren Reduktionen dieser Bestandteil ab der zweiten Reduktion bereits vorhanden ist, werden entsprechend nur noch Einträge (Menge E) hinzugefügt. Allgemein bildet jeder Eintrag ein Graphelement von g' auf die „tatsächliche" Id ab, die dieser durch die Auswertung der Terme erhalten sollte. Diese Ids sollen jedoch, gemäß der Umsetzungsskizze zu Beginn dieses Kapitels, nicht sofort gesetzt werden, sondern erst als eigenständiger Schritt nach dem die Transformation (genauer: jede mögliche Reduktion) durchgeführt ist. Jedes Graphelement j des Suchmusters von p, für durch den Bestandteil term in p ein Term definiert ist, ergibt einen Eintrag in E. Durch das Anwenden des Ersetzungsmorphismus hc bestimmt sich der Knoten, bestimmt sich das Graphelement, welches eine andere Id haben „sollte". Durch das Anwenden der Termauswertungsfunktion tef auf den zu j zugeordneten Term ergibt sich die „tatsächliche" Id. Dabei wird noch eine Variablenauflösung durchgeführt, um die Variablen jenes zugeordneten Terms durch die entsprechenden Graphelemente aus g zu ersetzen.

Definition 2.69: Id-Auflösung
Gegeben sei ein Graph $g \in$ GRAPH *für den der Bestandteil* idmap *definiert ist. Die* ***Id-Auflösung*** *ersetzt gemäß* $\underline{idmap}_g$ *die Ids in* g *und entfernt anschließend den Bestandteil* idmap, *und ist wie folgt definiert:*

$$resolveid(g) =_{def} \underline{idmap}_g(g) / \{\underline{idmap} \mapsto \underline{idmap}_g\}$$

Der Bestandteil idmap ist von der Struktur her ein Graphmorphismus und kann daher direkt auf g angewendet werden. Dabei werden alle Graphelemente die von $\underline{idmap}_g$ nicht abgebildet werden, bei der Anwendung des Graphmorphismus (als Funktion auf Graphen) auf sich selbst abgebildet.

2.5 Modelle

Die Syntax ist eine endlich große Darstellung der ggf. unendlich vielen Wörter einer Sprache. Mit Graphgrammatiken wurde in Kapitel 2.2.4 eine solche Darstellungsform eingeführt. Eine Graphgrammatik, bestehend aus Startgraph und Produktionsregeln, beschrieb die Wörter einer Sprache konstruktiv: Ausgehend von einem Wort der Sprache (dem Startgraph) ergab sich durch jede Anwendung einer Produktionsregel ein weiteres Wort der Sprache. Das Aufstellen einer solchen Syntax erfordert also das *Denken in Veränderungen*. In diesem Abschnitt wird eine alternative Definitionsform, die analytischen Sprachdefinition, eingeführt. Im Unterschied zu Graphgrammatiken geht es hierbei um das *Denken in Zuständen*. Anstatt anzugeben, wie sich Wörter durch Anwendung von Produktionsregeln aus anderen ergeben, wird bei einer analytischen Sprachdefinition die einzuhaltende Struktur der Wörter beschrieben (und im Rahmen des Wortproblems analysiert).

Die Modellierung wird in dieser Arbeit als eine Spezialform einer analytischen Sprachdefinition verstanden. Die Analyse wird durch einen Graph bestimmt. Die Idee dabei ist, dass das Modell, als Graph, einerseits den beschriebenen Wörtern strukturell ähnelt, so dass die Sprache mit in einer Art „Beispielen“ definiert werden kann. Andererseits abstrahieren diese „Beispiele“ von den konkreten Ids und Anzahlen beteiligter Graphelemente. Abbildung 2.39 zeigt ein Beispiel eines Modells m und der damit beschriebenen Sprache (Extension) in Form einiger Beispielgraphen die zu dieser gehören.

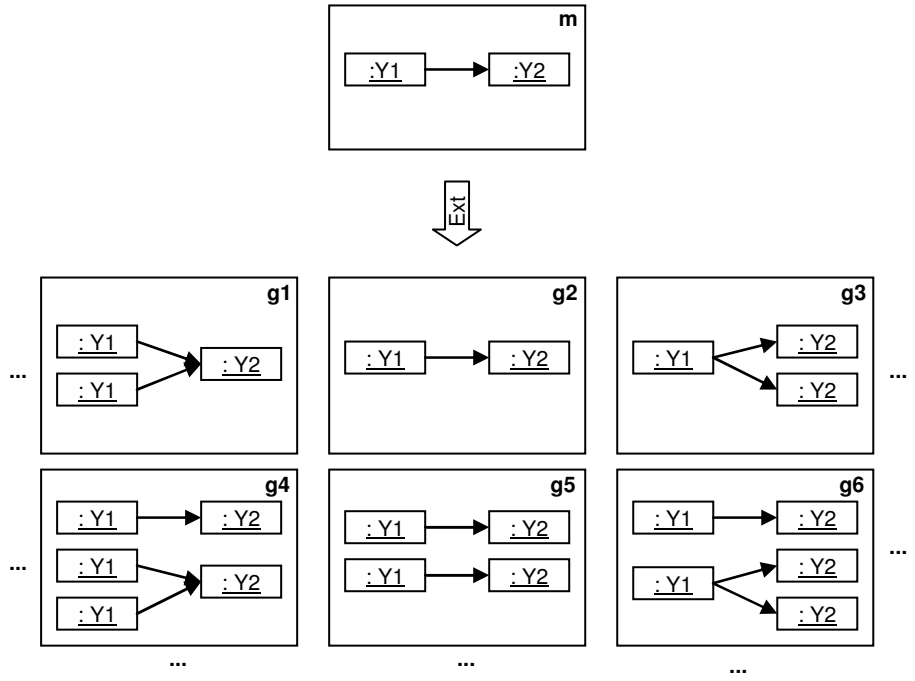


Abbildung 2.39 Extension eines Modells

Um Graphen, die als Modelle fungieren, von Graphen, die durch diese modelliert werden, besser unterscheiden zu können, wird die in Abbildung 2.40 gezeigte Darstellung für Modelle verwendet, die auch der von UML-Klassendiagrammen [47] entspricht. In diesem Zusammenhang entspricht ein Knoten eines Modells (bzw. dessen Typ) einer **Klasse**, und eine Kante eines Modells (bzw. deren Typ) einer **Assoziation**. Unabhängig von UML [47] werden jedoch Knoten eines Modells als **Modellknoten**, Kanten als **Modellkanten**, und allgemein Graphelemente als **Modellelemente** bezeichnet.

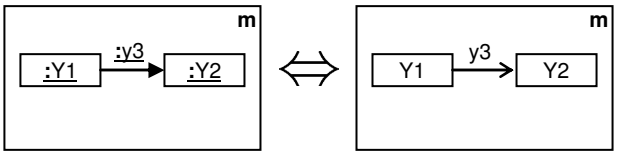


Abbildung 2.40 Notation für Modelle

Bei Umgangreichen Modellen wird es im Allgemeinen nicht möglich bzw. fachlich sinnvoll sein, Kanten überschneidungsfrei anzuordnen. Daher seien die in Abbildung 2.41 gezeigten Notationen für kompakte Darstellungen eingeführt. Die Kante vom Typ a in m verläuft von Y1 zu Y3. Dies lässt sich in der linken Darstellung daran erkennen, dass das Verbindungsstück zwischen Y1 und Y2 keine Pfeilspitzen besitzt. Es stellt somit keine Kante dar, sondern die ist die Verlängerung einer anderen. Analog lässt sich herauslesen, dass zwischen Y4 und Y6 eine Kante vom Typ b verläuft. Der Typ b bezieht sich hier auch auf die „unbenannte" Kante rechts „daneben", die zwischen Y5 und Y6 verläuft.

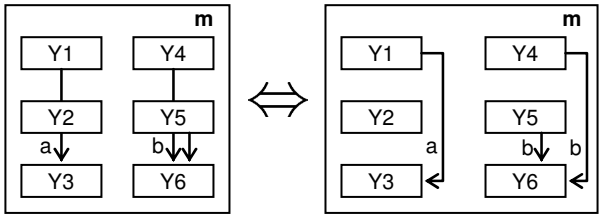


Abbildung 2.41 Notation zur kompakten Darstellung von Kanten in Modellen

Definition 2.70: Modell
Die Menge MODEL *aller* ***Modelle*** *ist wie folgt definiert:*

$$
\begin{aligned}
&MODEL =_{def} \{\, m \in GRAPH \mid \\
&\quad m(\underline{y})\ def. \wedge \\
&\quad wellformed<m> \wedge \\
&\quad [\forall\, t \in WB(\underline{y}_m): \\
&\qquad [\, |\{\, n \in nodes(m) \mid \underline{y}_m(n)=t \,\}| \leq 1 \,] \ \wedge \\
&\qquad [\, \forall\, n \in nodes(m): |\{\, e \in edges(m) \mid \underline{y}_m(e)=t \wedge \underline{s}_m(e)=n \,\}| \leq 1 \,] \wedge \\
&\qquad [\, \{\, n \in nodes(m) \mid \underline{y}_m(n)=t \,\} \cap \{\, e \in edges(m) \mid \underline{y}_m(e)=t \,\} = \emptyset] \\
&\quad]\,\}
\end{aligned}
$$

Ein Model ist ein getypter, wohlgeformter Graph für den folgende Bedingung gilt: Zu einem Typ t gibt es entweder maximal einen Knoten n, der diesen Typ hat, oder *pro* Knoten n maximal eine Kante e, die diesen Typ hat.

Definition 2.71: Instanz eines Modellknotens
Gegeben sei ein Modell m ∈ MODEL*, ein Modellknoten* nm ∈ nodes(m)*, ein Graph* g ∈ GRAPH *und ein Knoten* n ∈ nodes(g)*. Das Prädikat* nodeinstance<n,g,nm,m> *gibt an, ob* n *ein* ***Instanz*** *von* nm *bezüglich* g *und* m *ist, und ist wie folgt definiert:*

$$nodeinstance<n,g,nm,m> =_{def} \underline{y}_g(n)=\underline{y}_m(nm)$$

Ein Modellknoten n wird als Instanz eines Modellknotens nm bezeichnet, wenn dieser denselben Typ wie nm hat.

Definition 2.72: Instanz einer Modellkante
Gegeben sei ein Modell m ∈ MODEL*, eine Modellkante* em ∈ edges(m)*, ein Graph* g ∈ GRAPH *und eine Kante* e ∈ edges(g)*. Das Prädikat* edgeinstance<e,g,em,m> *gibt an, ob* e *eine* ***Instanz*** *von* em *bezüglich* g *und* m *ist, und ist wie folgt definiert:*

edgeinstance<e,g,em,m> $=_{def}$ ($\underline{y}_g(e)=\underline{y}_m(em)$) $\wedge$
nodeinstance<$\underline{s}_g(e)$,g,$\underline{s}_m(em)$,m> $\wedge$
nodeinstance<$\underline{t}_g(e)$,g,$\underline{t}_m(em)$,m>

Eine Kante e wird als Instanz einer Modellkante em bezeichnet, wenn diese denselben Typ wie em hat, und der Start- und Zielknoten von e Instanzen der Start- bzw. Zielknoten von em sind.

Definition 2.73: Modellkanten eines Modellknotens
Gegeben sei ein Modell m ∈ MODEL *und ein Modellknoten* nm ∈ nodes(m). *Die* ***Modellkanten*** *von* nm *in* m*, notiert mit* modeledges(nm,m)*, sind wie folgt definiert:*

modeledges(nm, m) $=_{def}$ { em∈edges(m) | $\underline{s}_m(em)$=nm }

Die zu einem Modellknoten nm zugehörigen Modellkanten sind diejenigen, die nm als Startknoten haben.

Definition 2.74: Instanz eines Modells
Gegeben sei ein Modell m ∈ MODEL *und ein Graph* g ∈ GRAPH. *Das Prädikat* instance<g,m> *gibt an ob* g *eine* ***Instanz*** *von* m *ist, und ist wie folgt definiert:*

instance<g,m> $=_{def}$ wellformed<g> $\wedge$
[$\forall$ n ∈ nodes(g):
[$\exists$ nm∈$\underline{ld}_m$:
nodeinstance<n,g,nm,m> $\wedge$
[$\forall$ e∈edges(g) mit $\underline{s}_g(e)$=n: [$\exists$em∈modeledges(nm,m): edgeinstance<e,g,em,m>]]
]
]

Ein Graph g wird dann als Instanz eines Modells m angesehen, wenn sich für jedes Graphelement in g ein Modellelement in m finden lässt, dessen Instanz es ist. Dabei wird durch Einsatz der Funktion modeledges berücksichtigt, dass ein Kantentyp bei verschiedenen Knotentypen in unterschiedlicher Weise verwendet werden kann.

Definition 2.75: Extension eines Modells, Konstrukt
Die durch ein Modell m ∈ MODEL *beschriebene Sprache, d.h. die* ***Extension*** *von* m*, wird mit* Ext(m) *notiert und ist die Menge alle Graphen die Instanz von* m *sind:*

Ext(m) $=_{def}$ { g ∈ GRAPH | instance<g,m> }

Die Menge der in einem Modell m *verwendeten Typen,* $\underline{y}_m$*, wird als die Menge der* ***Konstrukte*** *der so beschriebenen Sprache bezeichnet.*

Definition 2.76: Arbeiten mit Modellen
Gegeben sei ein Modell m ∈ MODEL*, ein Typ* y ∈ GETYPE *und ein Graph* g ∈ GRAPH *mit* instance<g,m>*. Die Menge aller Instanzen zum Typ* y *wird mit* instances(y,g) *notiert und ist wie folgt definiert:*

instances(y,g,m) $=_{def}$
$\bigcup$ n∈nodes(g) mit nodeinstance<n,g,any({nm∈nodes(m)|$\underline{y}_m$(nm)=y}),m>: { n }

Die Menge aller von n *durch Kanten des Typs* y *in* g *referenzierten Knoten, notiert mit* collect(n,y,g,m)*, ist wie folgt definiert:*

$$\text{collect}(n,y,g,m) =_{def} \bigcup e \in \text{edges}(g) \text{ mit } \underline{y}_g(e)=y \wedge \underline{s}_g(e)=n : \{ \underline{t}_g(e) \}$$

Die Menge aller Knoten die n *durch Kanten des Typs* y *in* g *referenzieren, notiert mit* collectors(n,y,g,m)*, ist wie folgt definiert:*

$$\text{collectors}(n,y,g,m) =_{def} \bigcup e \in \text{edges}(g) \text{ mit } \underline{y}_g(e)=y \wedge \underline{t}_g(e)=n : \{ \underline{s}_g(e) \}$$

Ist das verwendete Modell m *aus dem Kontext heraus klar, kann auf dessen Angabe bei der Verwendung der Funktionen* instances, collect *und* collectors *verzichtet werden.*

Falls anstelle eines Modellknotens nur dessen Typ zur Hand ist, kann mit der ersten Funktion die Menge aller dessen Instanzen erreicht werden. Die letzten beiden Funktionen decken die Fälle ab, in denen ein Knoten n mit einer Kante vom Typ y einen anderen Knoten referenziert oder selbst durch eine solche Kante referenziert wird. Dazu braucht das Modell m nicht ausgewertet zu werden da Kanten bei der Verfeinerung ihren Typ nicht ändern.

2.6 Erweiternde Konzepte für Modelle

In Kapitel 2.5 wurden erweiternde Konzepte für Graphen, Graphgrammatiken und Transformationen eingeführt. In diesem Abschnitt werden analog erweiternde Konzepte für Modelle vorgestellt.

2.6.1 Constraint

Bisher abstrahierte ein Modell vollständig z.B. von den Knotenanzahlen in einer Instanz. Mit Constraints können diese Einschränkungen verfeinert (d.h. beschränkt) werden. So kann die Anzahl beispielsweise entweder absolut auf einen konkreten Wert oder in Abhängigkeit von anderen existierenden Graphelementen formuliert werden. Wie die interne Syntax eines Constraints organisiert ist, bleibt dabei freigestellt. Insbesondere muss ein Constraint nicht selbst als Graph formuliert werden, sondern kann beispielsweise auch eine textuelle Sprache sein, die von sich aus höhere Sprachkonstrukte enthält, um die Untersuchung des vorgelegten Graphs kompakter darzustellen. OCL (Object Constraint Language) [48] ist eine solche Sprache zur Definition von Constraints, die von der OMG (Object Management Group) [49] veröffentlicht wird. Sie eignet sich insbesondere für graphbasierte Datenstrukturen und wird für das Weitere dieser Arbeit verwendet. Abbildung 2.42 zeigt ein Beispiel, in dem die beschriebene Sprache so eingeschränkt ist, dass jeder Knoten vom Typ Y1 nicht zu allen Knoten vom Typ Y2 über Kanten vom Typ a verbunden werden darf.

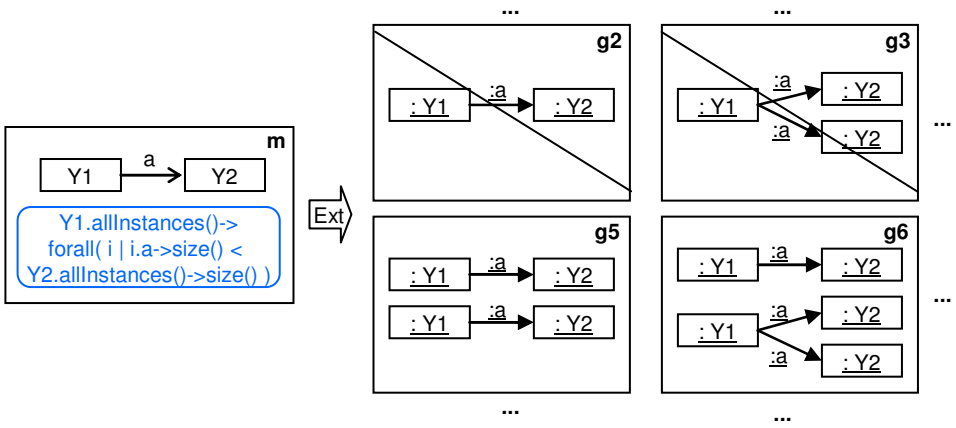


Abbildung 2.42 Modell mit Constraint

Definition 2.77: Constraint

Die Menge CONSTRAINT *aller* ***Constraints*** *ist wie folgt Spezifiziert:*

CONSTRAINT $\subseteq_{spec}$ STRING
CONSTRAINT $\supseteq_{spec}$ Menge der OCL-Constraints für Invarianten, ohne[10] OCL-Kontext

Die erste Spezifikationsbedingung besagt, dass ein Constraint ein String sein muss[11]. Darüber hinaus wird festgelegt, dass OCL-Constraints in der Menge CONSTRAINT vorkommen müssen. Für die entsprechende Syntax sei jedoch auf die Literatur [48] verwiesen.

Definition 2.78: Constraintauswertungsfunktion

[10] Ein OCL-Constraint mit OCL-Kontext wie z.B. `'context T inv: …'` kann stets auch ohne Kontxt durch `'T.allInstances()->forAll( self | … )'` dargestellt werden, und umgekehrt. Daher kann die Erörterung von OCL-Kontexten an dieser Stelle ausgespart werden.

[11] Dies wird, um vorzugreifen, in dieser Arbeit gebraucht um Constraints als Primitivwerte darzustellen, und so mit Produktionsregeln erstellen zu können.

*Die **Constraintauswertungsfunktion*** cef *ist eine Funktion die zu einem Contraintterm* ct ∈ CONSTRAINT *und einem Graph* g ∈ GRAPH *als Eingabe, einen Wahrheitswert* w ∈ BOOL *als Ausgabe liefert, der besagt ob* ct *für* g *erfüllt ist. Im Rahmen dieser Arbeit ist* cef *wie folgt spezifiziert:*

$\forall$ g ∈ GRAPH, ct ∈ CONSTRAINT:
cef(g,ct) $=_{spec}$ true gdw. ct für g unter der OCL-Semantik erfüllt ist

Für die Ausgestaltung der Funktion sei auf die Definition der OCL-Semantik [48] verwiesen. Zur Anwendbarkeit sind Knotentypen eines Graphen als Klassen, Knoten als Objekte, Kantentypen als Assoziationen und Kanten als Links zu betrachten. Dazu ist entsprechend die Verwendung des Konzepts der Typen notwendig, konkret um beispielsweise die in OCL häufig verwendeten Konstrukte zur Navigation über Assoziationen auch für Graphen anwenden zu können.

Definition 2.79: Modell mit Constraints
Für das Konzept der Constraints ist die Definition der Menge MODEL *wie folgt anzupassen:*

MODEL° $=_{def}$ { m ∈ MODEL | $\underline{Cnst}_m \subseteq$ CONSTRAINT }

Modelle werden um den Bestandteil $\underline{Cnst}$ erweitert der eine Menge von Constraints ist.

Definition 2.80: Instanz eines Modells
Für das Konzept der Constraints ist für einen Graph g ∈ GRAPH *und ein Modell* m ∈ MODEL *das Prädikat* instance<g,m> *wie folgt anzupassen:*

instance°<g,m> $=_{def}$ instance<g,m> $\wedge$ [$\forall$(ct,fp) ∈$\underline{Cnst}_m$: cef(g,ct)]

Damit ein Graph g als Instanz eines Modells m angesehen wird, müssen neben den bisherigen Bedingungen auch alle Constraints von m für g erfüllt sein.

2.6.2 Multiplizität

Multiplizitäten sind eine spezielle Form von Constraints die die Zahl der Instanzen von Modellknoten, in Relation zu einander, auf einfache Weise einschränken. Einfach bedeutet dabei, dass zur Einschränkung als Anzahl nur eine konkrete, natürliche Zahl verwendet werden kann. Diese lässt sich, wie Abbildung 2.43 (rechts) ebenfalls einfach darstellen (vgl. UML-Klassendiagramme). Eine Multiplizität besteht aus den zwei Teilen Minimum und Maximum, wobei jeweils neben einer natürlichen Zahl auch das Symbol * eingesetzt werden kann, was für „keine Beschränkung“ bzw. „beliebig“ steht.

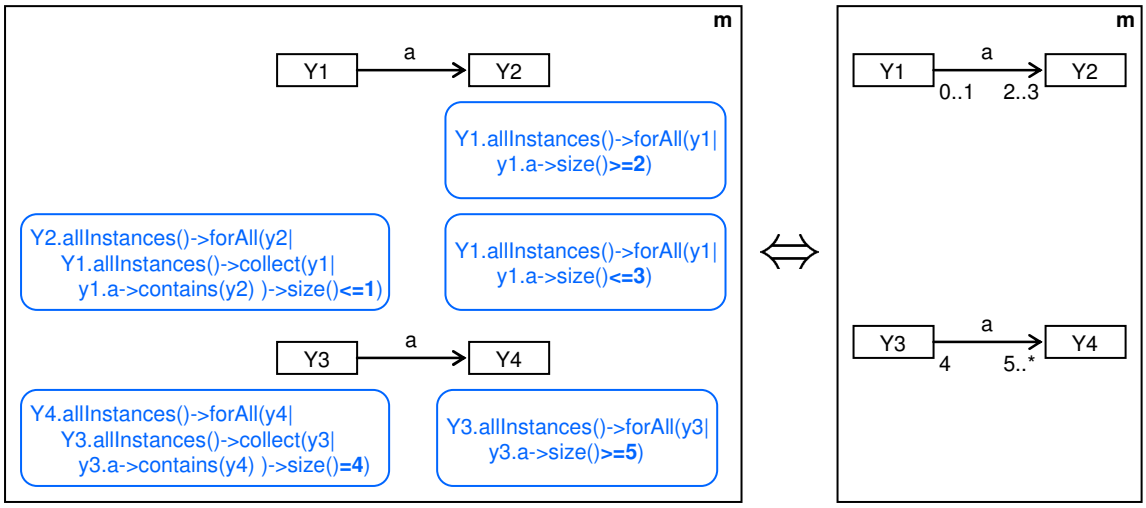


Abbildung 2.43 Notation für Multiplizitäten in Modellen

2.6.3 Vererbung

Das Konzept der Vererbung bietet zwei Wirkungen: Einerseits, wie Abbildung 2.44 zeigt, können Modelle kompakter notiert werden, indem Klassen mit ähnlichen umgebenden Strukturen als einander erweiternd definiert werden. Konkret haben beide Klassen Y1 und Y4 ausgehende Kanten mit Typen a und b. Da Y4 sich von Y1 nur durch die zusätzlich ausgehende Kante c unterscheidet, kann Y4 als Erweiterung von Y1 angesehen werden. Im Sprachgebrauch **erbt** Y4 von Y1. In dem Modell ist dann für Y4 nur noch die gegenüber Y1 zusätzliche Kante darzustellen.

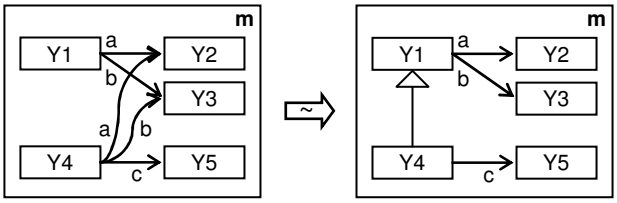


Abbildung 2.44 Vererbung als Mittel zur kompakten Notation von Modellen

Andererseits ist die Vererbung inhärent mit der Teilmengenbeziehung verbunden, die besagt, dass alle Instanzen der erbenden Klasse (hier Y4), automatisch auch als Instanzen der vererbenden Klasse (hier Y1) anzusehen sind[12]. Daraus leitet sich die Möglichkeit ab, bei der Vererbung auch den Typ des Zielknotens einer Kante weiter einzuschränken. Abbildung 2.45 zeigt die Notation an einem Beispiel, bei dem die Klasse Y4 von Y1 und Y5 von Y2 erbt. Während jedem Objekt vom Typ Y1 über eine Kante vom Typ a ein Objekt vom Typ Y2 zugeordnet werden darf, wird durch die nochmalige Definition einer Kante vom Typ a in der von Y1 erbenden Klasse Y4, diesmal jedoch mit Y5 als Zielknoten, ausgedrückt, dass zu Objekten vom Typ Y4 nur solche Instanzen vom Typ Y2 zuordenbar sind, die auch vom Typ Y5 sind.

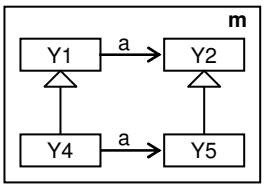


Abbildung 2.45 Implizite Verfeinerung von Kanten

[12] Sofern dieser Effekt nicht gewünscht ist, kann die Vererbung nicht für eine kompaktere Notation (wie in Abbildung 2.44 dargestellt) verwendet werden.

Für die Formalisierung werden Vererbungsbeziehungen im Modell durch den neuen Bestandteil inherit abgelegt der sich an [50] orientiert. Der Zusammenhang zur Notation wird in Abbildung 2.46 gezeigt.

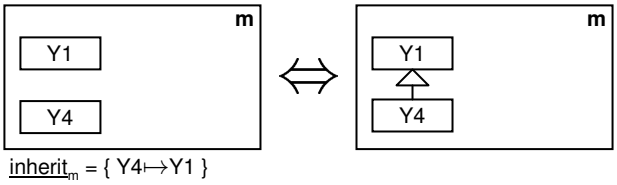


Abbildung 2.46 Ablage der Vererbungsbeziehungen als Bestandteil eines Modells

Definition 2.81: Modell mit Vererbung
Für das Konzept der Vererbung ist die Definition der Menge MODEL *wie folgt anzupassen:*

$$
\begin{aligned}
&MODEL^{\circ} = \{\ m \in MODEL \mid \\
&\quad \underline{inherit}_m : WB(\underline{y}_m) \rightarrow WB(\underline{y}_m) \ \wedge \\
&\quad [\ \forall\ y \in WB(\underline{y}_m) : (y,y) \notin transhull(\underline{inherit}_m)\ ] \ \wedge \\
&\quad [\ (DB(\underline{inherit}_m) \cup WB(\underline{inherit}_m)) \subseteq [\bigcup n \in nodes(m) : \{\underline{y}_n\}]\ ] \ \wedge \\
&\quad [\ \forall\ em1,em2 \in edges(m) \text{ mit } \underline{y}_m(em1)=\underline{y}_m(em2): \\
&\qquad (\underline{y}_m(\underline{s}_m(em1)),\underline{y}_m(\underline{s}_m(em2))) \in transhull(\underline{inherit}_m) \Longrightarrow \\
&\qquad (\underline{y}_m(\underline{t}_m(em1)),\underline{y}_m(\underline{t}_m(em2))) \in transhull(\underline{inherit}_m) \\
&\quad]\ \}
\end{aligned}
$$

Ein Modell wird um den Bestandteil inherit erweitert. Dieser setzt Typen, im Sinne der Vererbungsbeziehung bei UML-Klassendiagrammen, zu einander in Bezug: Ein $(y1,y2) \in \underline{inherit}_m$ besagt, dass der Typ y1 vom Typ y2 erbt. Die Vererbungsbeziehung muss zyklenfrei sein. Vererbungen können nur zwischen Typen von Modellknoten festgelegt werden. Die Vererbung zwischen Modellkanten wird implizit gehandhabt: Wenn zwei Modellkanten em1 und em2 zu zwei Modellknoten $\underline{s}_m(em1)$ und $\underline{s}_m(em2)$ gehören, die in einer Vererbungsbeziehung zu einander stehen, wird em1 als Erbe (bzw. als Verfeinerung) von em2 angesehen. In einem solche Fall müssen die Zielmodellknoten, hier $\underline{t}_m(em1)$ und $\underline{t}_m(em2)$, ebenfalls in Vererbungsbeziehung stehen.

Definition 2.82: Instanz eines Modellknotens bei Vererbung
Für das Konzept der Vererbung ist für ein Modell $m \in$ MODEL*, ein Modellknoten* $nm \in nodes(m)$*, ein Graph* $g \in$ GRAPH *und einen Knoten* $n \in nodes(g)$ *die Definition des Prädikats* nodeinstance<n,g,nm,m> *wie folgt anzupassen:*

$$nodeinstance^{\circ}\langle n,g,nm,m\rangle =_{def} (\underline{y}_m(n),\underline{y}_m(nm)) \in transhull(\underline{inherit}_m)$$

Durch die Anpassung wird ein Knoten n auch dann als Instanz eines Modellknotens nm auch dann angesehen, wenn er zwar nicht denselben Typ hat, aber einen, der transitiv davon erbt.

Definition 2.83: Modellkanten eines Modellknotens bei Vererbung
Für das Konzept der Vererbung ist für ein Modell $m \in$ MODEL*, ein Modellknoten* $nm \in nodes(m)$*, ein Graph* $g \in$ GRAPH *und einen Knoten* $n \in nodes(g)$ *die Definition der Menge* Modeledges *wie folgt anzupassen:*

$$
\begin{aligned}
&modeledges^{\circ}(\ nm,\ m\ ) =_{def}\ modeledges(nm,m) \cup H(nm,n) \\
&\text{wobei} \\
&\quad parent(nm,m) = any(\{pm \in nodes(m) \mid \underline{y}_m(pm)=\underline{inherit}_m(\underline{y}_m(nm))\}) \\
&\quad H(nm,m) = \{\ em \in modeledges^{\circ}(parent(nm,m),m) \mid
\end{aligned}
$$

$\forall$ em2$\in$modeledges(nm,m) : $\underline{y}_m$(em2)$\neq\underline{y}_m$(em)] }

Die Menge der Modellkanten zu einem Modellknoten nm umfasst, im Gegensatz zur ursprünglichen Definition, nicht mehr alle von nm ausgehenden Modellkanten. Grund dafür ist, dass Modellkanten verfeinert werden können – so dass für einen Modellknoten maximal eine Modellkante pro Typ verwendet werden darf. Daher wird, ausgehend vom Modellknoten nm und mit Blick auf die „über“ nm liegenden Vererbungshierarchien, nur die jeweils erste Modellkante pro Typ in die Ergebnismenge aufgenommen (siehe Abbildung 2.47).

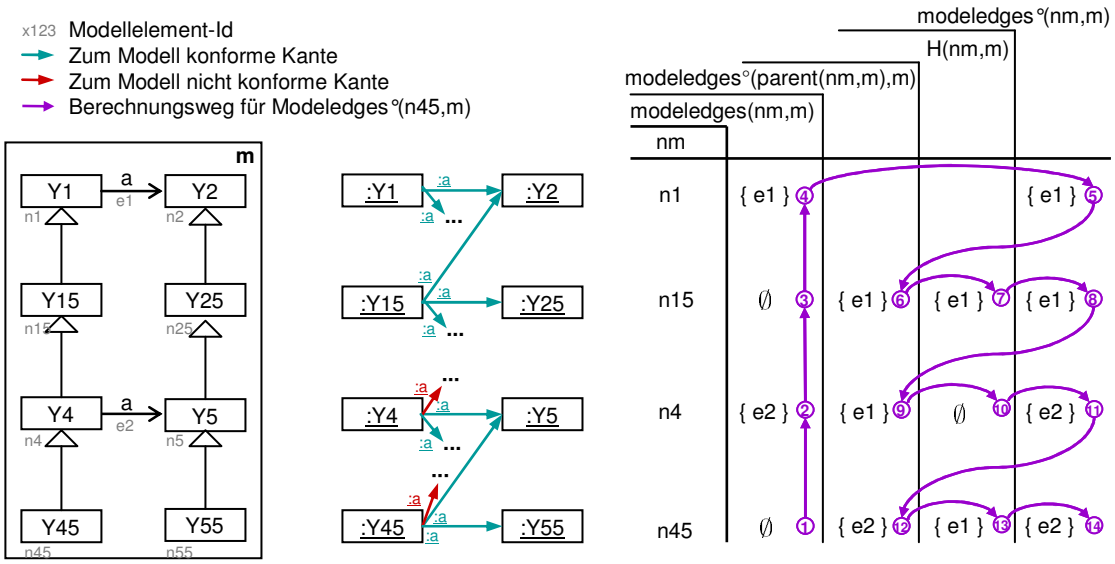
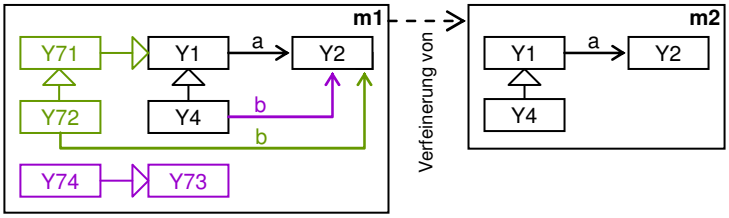


Abbildung 2.47 Verfeinerung von Kanten

Definition 2.84: Verfeinerung von Modellen
Gegeben seien zwei Modelle m1, m2 ∈ MODEL. *Das Prädikat* refinement<m1,m2> *gibt an, ob das Modell* m1 *eine Verfeinerung von Modell* m2 *ist, und ist wie folgt definiert:*

refinement<m1,m2> $=_{def}$
[$\forall$ b$\in$DB(m2) : $b_{m2} \subseteq b_{m1}$] $\wedge$
[$\forall$ nm1$\in$nodes(m2)/nodes(m1):
[$\exists$ nm2$\in$nodes(m2):
($\underline{y}_{m1}$(nm1),$\underline{y}_{m2}$(nm2)) $\in$ transhull($\underline{inherit}_{m1}$)]] $\wedge$
[$\forall$ em1 $\in$ edges(m2)/edges(m1) :
$\underline{s}_{m1}$(em1) $\notin$ nodes(m1)]

Die erste Bedingung besagt, dass sämtliche Bestandteile aus m2 auch in dem verfeinerten Modell m1 jeweils im vollen Umfang vorhanden sein müssen – dazu gehören sämtliche Graphelemente, Start- und Zielknotenzuordnungen, Typzuordnungen und ggf. etwaige Erweiterungen wie z.B. der inherit-Bestandteil. Im konkreten Beispiel in Abbildung 2.48 sind das alle schwarz gefärbten Elemente. Die zweite Bedingung besagt, dass die in m1 gegenüber m2 neu hinzugefügten Knoten, über die Vererbungshierarchie letztlich von einem Knoten aus m2 erben müssen. In Abbildung 2.48 erbt Y71 direkt, und Y72 indirekt von einem Knoten aus m2. Dies trifft für Y73 und Y74 nicht zu, weshalb diese eine ungültige Verfeinerung bilden würden. Die dritte Bedingung besagt, dass in m1 gegenüber m2 neu hinzugefügte Kanten nicht an in m2 bereits existierende Knoten hinzugefügt werden dürfen – sondern nur an die in m1 ebenfalls hinzugefügten. In Abbildung 2.48 betrifft das die Kante b, die nicht zu Y4 hinzugefügt werden darf. Dieses Verständnis entspricht der Vererbung in verbreiteten Programmiersprachen wie Java oder C^{++} (vgl. [62]), wenn die Modelle m1 und m2 als zwei Klassen verstanden werden, die in einer Vererbungsbeziehung zueinander stehen.



grün : Elemente bilden gültige Verfeinerung
violett : Elemente bilden ungültige Verfeinerung

Abbildung 2.48 Verfeinerung von Modellen

2.6.4 Abstrakter Typ

Abstrakte Typen sind eine spezielle Form von Constraints, die die Zahl der Instanzen von Modellknoten auf einfache Weise einschränken. Als Anwendungsbeispiel sei folgeneds Szenario betrachtet: Instanzen eines Typs Y2 sollen Instanzen eines Typs Y1 referenzieren können – allerdings nur solche, die *entweder* eine eingehende *oder* eine ausgehende Kante zu einer Instanz vom Typ Y3 besitzen. Abbildung 2.49 zeigt wie dieses Szenario unter Verwendung von abstrakten Typen beschrieben werden kann: Von Y1 werden die beiden Typen Y1in und Y1out abgeleitet, die die beiden Fälle abdecken. Y1 wird als abstrakt definiert. Dadurch dürfen in einer Instanz des Modells keine Knoteninstanzen existieren, die Y1 als direkten Typ haben. Instanzen können dann nur noch von erbenden Typen gebildet werden (sofern diese nicht wiederum abstrakt sind). Zur Notation werden abstrakte Typen kursiv dargestellt (Abbildung 2.49 rechts).

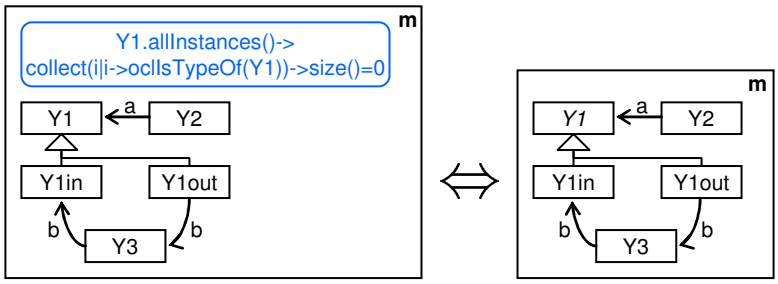


Abbildung 2.49 Modelle mit abstrakten Typen

2.6.5 Primitivwert

Wie bei Graphen seien Primitivwerte auch bei Modellen über Ids transportiert. Die dazu einzig notwendige Anpassung besteht darin, die Wirkung reservierter Typen (wie z.B. den Typ Zeichenkette) von der Graphgrammatikwelt in die Welt der analytischen Sprachdefinitionen zu übertragen. Konkret bedeutet dies die Anpassung des Instanzbegriffes (als Gegenstück zur Anpassung der Patternmatchingfunktion bei Graphgrammatiken).

Definition 2.85: Instanz eines Modells bei Primitivwerten
Für das Konzept der Primitivwerte ist für einen Graph g ∈ GRAPH *und ein Modell* m ∈ MODEL *das Prädikat* instance<g,m> *wie folgt anzupassen:*

$$\text{instance}^{\circ}\langle g,m\rangle =_{def} \text{instance}\langle g,m\rangle \wedge [\forall i \in \underline{Id}_g \text{ mit } \underline{y}_g(i)=\text{Zeichenkette} : i \in \text{STRING}]$$

Damit ein Graph g als Instanz eines Modells m angesehen wird, muss, neben den bisherigen Bedingungen, für jedes Graphelement i mit Typ Zeichenkette gelten, dass i aus STRING ist.

2.6.6 Attribut

Attribute werden in dieser Arbeit der Einfachheit halber als eine besondere Situation bezüglich der Multiplizitäten zwischen den Modellknoten eines Modells verstanden. Abbildung 2.50 zeigt diese Vorstellung an einem konkreten Beispiel.

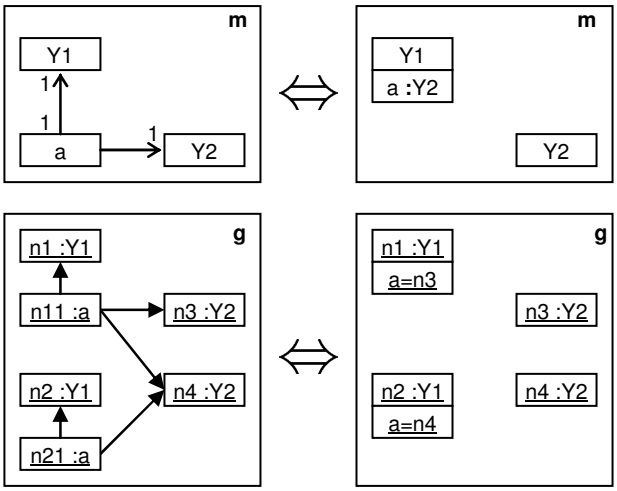


Abbildung 2.50 Attribute in Modellen am Beispiel

Sofern zwischen einem Modellknoten Y1 und einem Modellknoten a eine 1:1-Multiplizitätsbeziehung herrscht, wird der Modellknoten a als Attribut von Y1 angesehen, und wird gemäß der Notation von UML-Klassendiagrammen als „Teil“ von Y1 dargestellt (Abbildung 2.50, oben rechts). Der Typ des Attributs (direkt nach dem Namen des Attributs und durch einen Doppelpunkt getrennt angegeben) ergibt sich aus der „anderen“, von a abgehenden Modellkante zum Modellknoten Y2. Diese Modellkante muss als Voraussetzung die Multiplizität 1 für den Zielknoten besitzen. Sollte dies nicht der Fall sein, oder a darüber hinaus noch weitere abgehende Modellkanten besitzen, wird nicht von einem Attribut gesprochen.

Auf Ebene der Instanzen (d.h. Graphen, vgl. Abbildung 2.50, unten) werden Wertbelegungen für Attribute ebenfalls als Teil des besitzenden Graphelements dargestellt. Um bei mehreren Attributen darzustellen, zu welchem Attribut ein Wert zugeordnet ist, wird eine Kombination aus Attribut und Wert, z.B. a=n3, abgelegt, wobei n3 der Wert ist. Eine solche Kombination wird als (mit einem Wert belegter) **Slot** bezeichnet.

Das in dieser Arbeit verwendete Verständnis von Attributen steht im Gegensatz zu dem in der Literatur üblichen Verständnis attribuierter Graphen (vgl. [51,52]): Attribute (bzw. Slots) werden hier syntaktisch nicht explizit erfasst, sondern als syntaktischer Zucker (vgl. [53]) verstanden, und damit nur als eine Notation gehandhabt. Dies vereinfacht den Einsatz der inkrementellen Transformation auf mit Modellen beschriebene, attribuierte Graphen, da die in Kapitel 2.3.1 definierte Funktion unmittelbar eingesetzt werden kann – ohne sie für die Behandlung von Attributen (bzw. Slots) anpassen zu müssen.

2.7 Dynamische Transformationen

Das Szenario einer Transformation (Kapitel 2.2.4) wurde eingeführt als das erschöpfende Reduzieren eines frei wählbaren Graphen g mit einer zuvor festgelegten Menge von Produktionsregeln. Offensichtlich können diese Produktionsregeln durch die Wahl von g nicht verändert werden. Die Produktionsregeln können aber so gewählt sein, dass sie Teile von g als auszuführende Produktionsregeln interpretieren und somit die Menge der Produktionsregel *scheinbar* verändern. Mit der Wahl von g kann somit – effektiv gesehen – auch die auszuführende Transformation beeinflusst werden. Wie dies konkret aussieht, wird in Kapitel 2.7.1 an einem Beispiel gezeigt. Kapitel 2.7.2 beschreibt die Virtualisierung als eine Interpretation, die durch eine Schnittstelle „abgesichert". Dafür werden mit einem Modell Typen reserviert, mit denen Produktionsregeln als Teil eines Graphen g abgelegt werden können, an denen sich die Ausführungssemantik orientiert.

2.7.1 Interpretation

In Abbildung 2.51 ist das Szenario einer „nicht-interpretierenden" Transformation an einem Beispiel dargestellt: Egal wie g gewählt wird, die Produktionsregeln in tr können nicht beeinflusst werden. Stets werden in dem gewählten Graph zwei aufeinander folgende, gleichgerichtete Kanten entfernt.

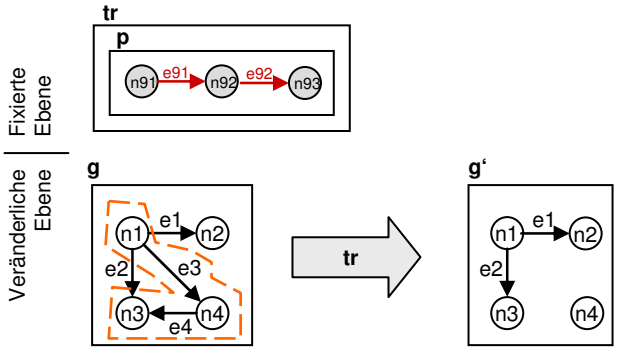


Abbildung 2.51 Nicht-interpretierende Transformation

Abbildung 2.52 zeigt wie die Produktionsregel p (aus dem Beispiel in Abbildung 2.51) als Teil des Graphen abgelegt werden kann. Um die so in g hinzugefügten Graphelemente von den bisherigen „Nutzdaten" (Knoten n1–n4, Kanten e1–e4) und auch untereinander zu unterscheiden, werden in diesem Beispiel der Einfachheit[13] halber getypte Graphen mit entsprechenden Typen (nd, ed, dl, src, trg) verwendet. In den Produktionsregeln der Transformation tr kommt die so abgelegte Produktionsregel nicht mehr vor. Stattdessen werden elf Produktionsregeln (siehe Abbildung 2.54) eingeführt die in einem Graph g virtuell abgelegten Produktionsregeln interpretieren und entsprechend ausführen. Abbildung 2.53 zeigt, wie mit der Wahl von g das Transformationsverhalten beeinflusst werden kann. Die erste abgelegte Produktionsregel führt zum Entfernen sämtlicher[14] Kanten. Die zweite Produktionsregel führt zum Entfernen der ersten Kante in einer Zweierkette, während die letzte Produktionsregel die zweite Kante in einer solchen Kette entfernt. Darüber hinaus kann die durch die Produktionsregeln in Abbildung 2.54 definierte Transformation alle solchen abgelegten Regeln verarbeiten, die eine beliebig lange Kette aus Kanten suchen, und daraus beliebig viele löschen.

[13] Ist im Allgemeinen nicht notwendig, da Typen als individuelle Graphstrukturen kodiert werden könnten.
[14] Die Transformationsregeln werden durch Domänen geschützt

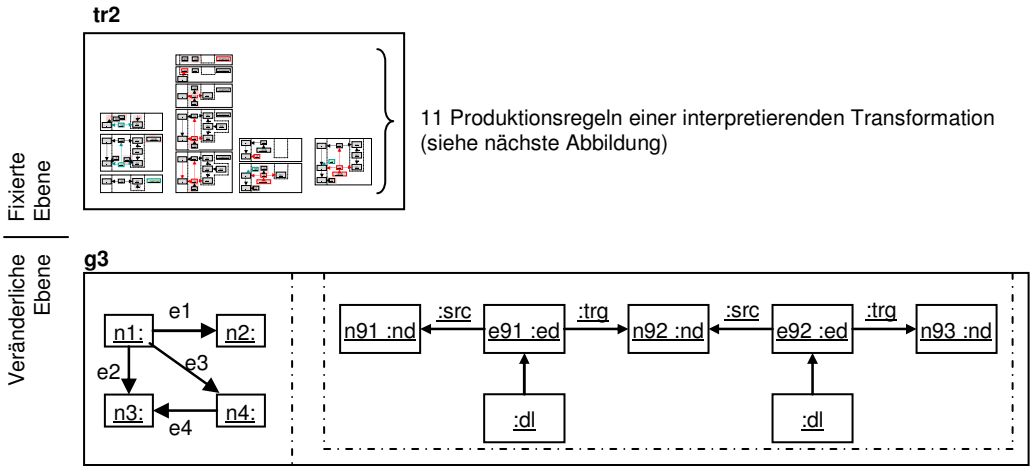


Abbildung 2.52 Interpretierende Transformation

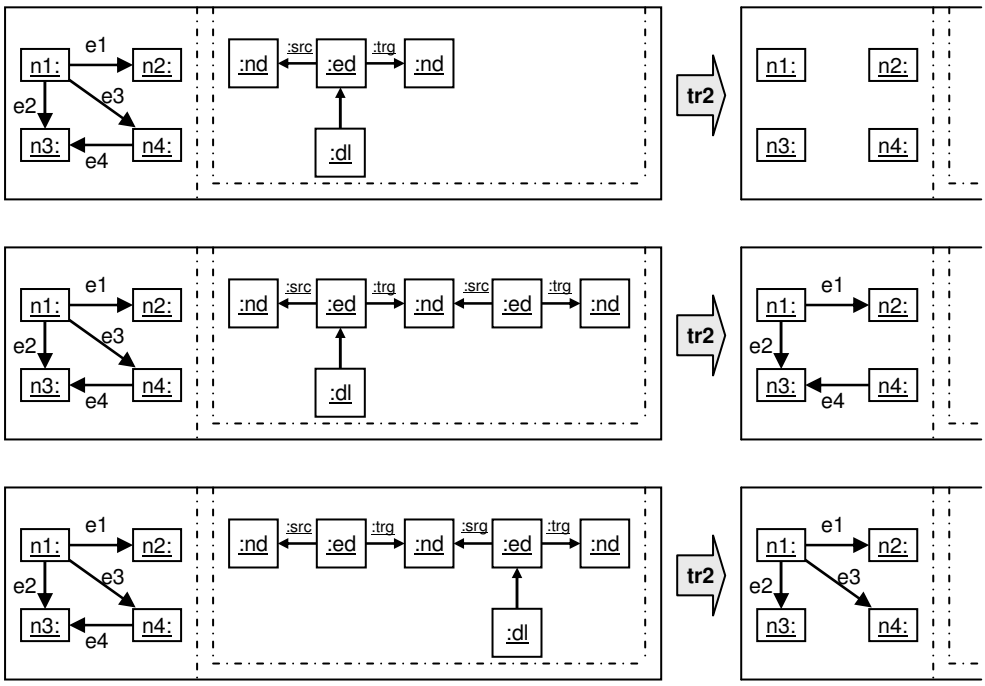


Abbildung 2.53 Beispiele zur Ausführung einer interpretierenden Transformation

Abbildung 2.54 zeigt die konkreten Produktionsregeln der interpretierenden Transformation aus dem Beispiel aus Abbildung 2.52. Die Produktionsregeln p1 – p3 bilden das Patternmatching nach. Im Erfolgsfall wird ein Knoten vom Typ matched erstellt. Die Elemente vom Typ m repräsentieren dabei den gefundenen Redexmorphismus. Durch p4 – p8 wird die Ersetzung vorgenommen. Je nachdem ob ein Knoten vom Typ dl vorhanden ist, wird eine Kante gelöscht, oder beibehalten. Die Produktionsregeln p9 – p11 gehören ebenfalls zum Patternmatching. Sie behandelt das allgemeine Backtracking, falls die Produktionsregeln p1 – p4 beim Aufbau einer Belegung in eine „Sackgasse“ geraten. Um die Zahl der Produktionsregeln der Transformation minimal[15] zu halten, wird für eine korrekte[16] Anwendung vorausgesetzt, dass die Nutzdaten von g keine Zyklen aufweisen. Auch wird der Problempunkt ausgeblendet, dass die Transformation so nicht terminiert, da das Patternmatching auch dann fortgesetzt wird, wenn keine Redexe mehr vorhanden sind[17].

[15] Unter dem Aspekt eines dennoch aussagekräftigen Beispiels
[16] Sofern Zyklen vorhanden sind, werden Kanten-Ketten die durch einen Zyklus verlaufen nicht erkannt.
[17] Ein einfacher Weg bestünde im Erzeugen eines Abbruch-Tokens bei scheitern eins Patternmatchingdurchlaufs. Es bliebe jedoch ein „Artefakt“ übrig welches nach der Transformation gesondert zu entfernen wäre.

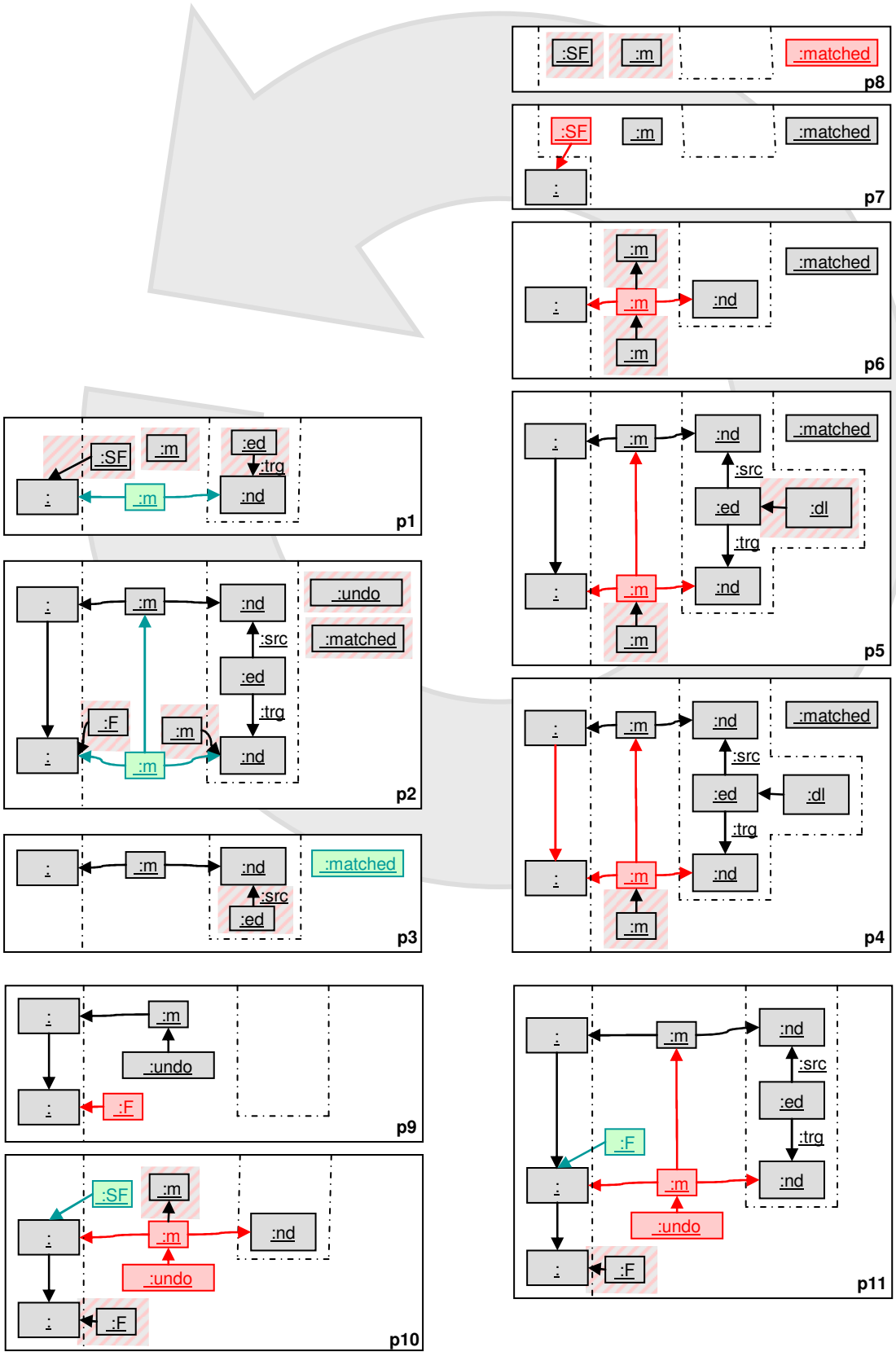


Abbildung 2.54 Beispiel einer interpretierenden Transformation

2.7.2 Virtualisierung

Das vorangegangene Beispiel aus Kapitel 2.7.1 hat gezeigt, dass die scheinbar wirkenden Produktionsregeln durch eine geschickte Wahl der festen Produktionsregeln beeinflusst werden können – ohne die Berechnungsvorschrift für die Transformationsanwendung zu verändern. Erreicht wurde dies, indem die festen Produktionsregeln jene Berechnungsvorschrift nachbildeten. Bereits für das kleine, eingeschränkte Beispiel waren dazu 6 Produktionsregeln allein für das Patternmatching nötig. Um im weiteren Verlauf mit beliebigen virtualisierten Produktionsregeln hantieren zu können, ohne den zur Auswertung notwendigen Satz fester Produktionsregeln in dann noch größerem Umfang spezifizieren zu müssen, wird das Konzept der Virtualisierung eingeführt.

Die Virtualisierung basiert auf einer Erweiterung des Ablaufs einer Transformationsanwendung, bei der in die Berechnung des Transformationsergebnisses eine Extraktionsfunktion (mit devirtualize notiert) integriert wird. Diese extrahiert aus dem zu transformierenden Graphen die anzuwendenden Produktionsregeln und fügt diese bei der Transformation zu dem verwendeten festen Satz von Produktionsregeln hinzu. Grundlage für die Extraktionsfunktion ist dabei die Festlegung der Struktur, in der Produktionsregeln in einem Graph abgelegt sind. Abbildung 2.55 zeigt das dazu verwendete Modell. Abbildung 2.56 zeigt ein Beispiel einer Instanz dieses Modells (rechts) und der dadurch beschriebenen Produktionsregel (links).

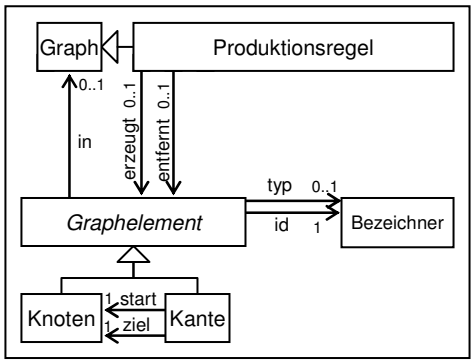


Abbildung 2.55 Modell für Virtualisierung von Graphen und Produktionsregeln

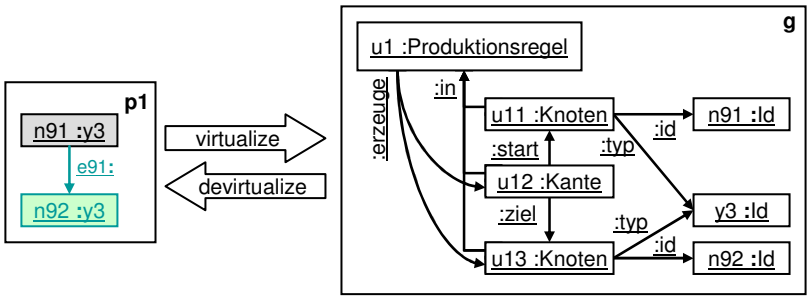


Abbildung 2.56 Beispiel für Virtualisierung und Devirtualisierung

Zur übersichtlicheren Darstellung wird die in Abbildung 2.57 gezeigte Notation vereinbart, mit der eine in einem Graph virtualisiert abgelegter Produktionsregel dargestellt wird. Um die Koexistenz mit anderen Graphelementen (die keine Produktionsregel virtualisieren) zu zeigen, wurde zum Graph g aus Abbildung 2.56 der Knoten n90 mit Typ y1 hinzugefügt. Die Notation entspricht im Wesentlichen der in Abbildung 2.56 virtualisierten Produktionsregel p1, mit dem Unterschied, dass es für die Produktionsregel einen getypten Rahmen gibt, und die Elemente der Produktionsregel innerhalb dieses Rahmens und mit gepunkteten Linien dargestellt sind.

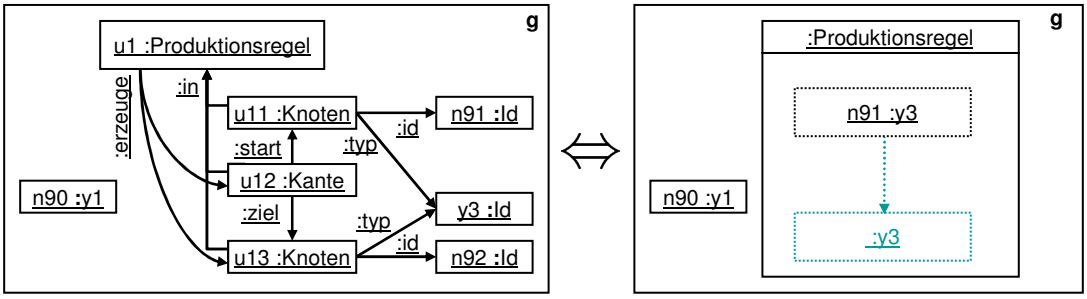
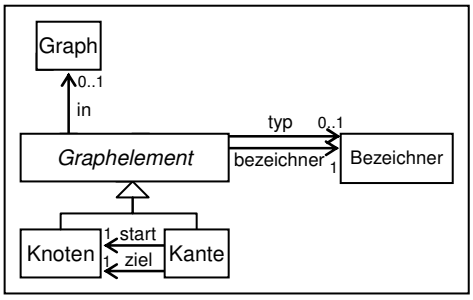


Abbildung 2.57 Notation für Graphen, die virtualisierte Produktionsregeln enthalten

Da Produktionsregeln immer auch Graphen sind (wegen PRODRULE $\subseteq$ GRAPH), lassen sich mit der Virtualisierung von Produktionsregeln auch einfach nur Graphen virtualisieren. Dies spiegelt sich in dem Virtualisierungsmodell (Abbildung 2.55) wieder, in dem das Konstrukt der Produktionsregel vom Konstrukt Graph erbt. Entsprechend wird die De- und Virtualisierung zunächst für Graphen formalisiert, um anschließend die für die Produktionsregeln darauf aufbauend anzugeben. Der praktische Nutzen für eine Virtualisierung von Graphen ergibt sich jedoch erst im nachfolgenden Kapitel 2.8.

Definition 2.86: Virtualisierungsmodell für Graphen
*Das **Virtualisierungsmodell*** GraphVirtModel $\in$ MODEL *ist ein ausgezeichnetes Modell für die Virtualisierung von Graphen, und ist wie folgt definiert:*

GraphVirtModel $=_{def}$



Das GraphVirtModel ist ein ausgezeichnetes Modell um Graphen als Graphen darzustellen. Dabei können mit diesem Modell mehrere Graphen dargestellt werden.

Definition 2.87: Devirtualisierungsfunktion für Graphen
*Die **Devirtualisierungsfunktion*** devirtualizegraphs *nimmt eine Instanz des Graphvirtualisierungsmodells, also ein Element aus* { g $\in$ GRAPH | instance<g,GraphVirtModel> }*, als Eingabe, und extrahiert daraus alle darin virtualisierten Graphen als „echte“ Elemente von* GRAPH*, und ist wie folgt definiert:*

$$\text{devirtualizegraphs}(g) =_{def} \bigcup \text{gid} \in \text{instances}(\text{Graph},g) : \{ \text{devirtualizegraph}(g, \text{gid}) \}$$

Die Funktion devirtualizegraph *extrahiert einen einzelnen Graph, den mit der als Parameter angegebenen Id* gid*, aus einer Instanz des Virtualisierungsmodells* g*, und ist wie folgt definiert:*

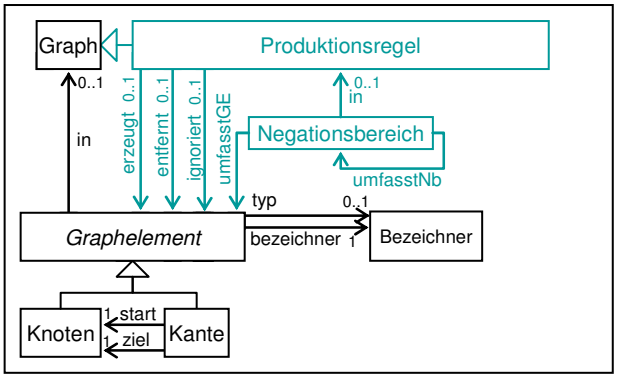
$$\text{devirtualizegraphs}(g,\text{gid}) =_{def} h(g) \text{ mit}$$
$$g' = \{ \underline{\text{Id}} \mapsto \text{Id}', \underline{s} \mapsto s', \underline{t} \mapsto t', \underline{y} \mapsto y' \} \text{ mit}$$
$$\text{Id}' = (\text{instances}(\text{Knoten},g) \cup \text{instances}(\text{Kante},g)) \cap \text{collectors}(\text{gid},\text{in},g)$$
$$s' = \bigcup k \in \text{instances}(\text{start},g) : \{ (\underline{s}_g(k),\underline{t}_g(k)) \mid \underline{s}_g(k) \in \text{Id}' \}$$

$t' = \bigcup k \in instances(ziel,g) : \{ (\underline{s}_g(k),\underline{t}_g(k)) \mid \underline{s}_g(k) \in Id' \}$
$y' = \bigcup k \in instances(typ,g) : \{ (\underline{s}_g(k),\underline{t}_g(k)) \mid \underline{s}_g(k) \in Id' \}$
$h = \bigcup k \in instances(id,g) : \{ (\underline{s}_g(k),\underline{t}_g(k)) \mid \underline{s}_g(k) \in Id' \}$

Die Devirtualisierung eines virtualisierten und mit gid identifizierten Graphen läuft wie folgt ab: Zunächst werden mit Id' alle Knoten und Kanten extrahiert, die zu gid gehören – was sich anhand der Kanten vom Typ in feststellen lässt. Anschließend werden aus den Kanten der Typen start, ziel und typ Graphbestandteile abgeleitet, wobei wiederum darauf geachtet wird, dass sie zum mit gid identifizierten Graphen gehören. Das so entstehende Gebilde g' ist damit ein „echter" Graph, d.h. es gilt g' ∈ GRAPH. Allerdings entsprechen die Ids seiner Graphelemente noch nicht mit dem ursprünglich virtualisierten überein. Dazu müssen die Kanten vom Typ id ausgewertet und angewendet werden. Hierzu eignet sich die Konstruktion und Anwendung eines Graphmorphismus h, der sowohl die Graphelemente, als auch die restlichen Bestandteile $\underline{s}$, $\underline{t}$ und $\underline{y}$ auf die richtigen Ids setzt.

Definition 2.88: Virtualisierungsmodell für Produktionsregeln
*Das **Modell*** ProdVirtModel ∈ MODEL ***zur Virtualisierung von Produktionsregeln*** *ist wie folgt definiert:*

ProdVirtModel $=_{def}$



Das Modell ProdVirtModel erweitert das GraphVirtModel um die grün gefärbten Konstrukte. Domänen und Operationen werden nicht virtualisiert abgebildet, da dies im Weiteren nicht benötigt wird.

Definition 2.89: Devirtualisierungsfunktion für Produktionsregeln
*Die **Devirtualisierungsfunktion*** devirtualizeprods *nimmt eine Instanz des Virtualisierungsmodells, also ein Element aus* { g ∈ GRAPH | instance<g,ProdVirtModel> }*, als Eingabe, und extrahiert daraus die virtualisierten Produktionsregeln als „echte" Elemente von* PRODRULE*, und ist wie folgt definiert:*

$devirtualizeprods(g) =_{def} \bigcup pid \in instances(Produktionsregel,g) : \{ devirtualizeprod(g, pid) \}$

Die Funktion devirtualizeprod *extrahiert eine einzelne Produktionsregel, die mit der als Parameter angegebenen Id* pid*, aus einer Instanz des Virtualisierungsmodells* g*, und ist wie folgt definiert:*

$devirtualizeprod(g,pid) =_{def} devirtualizegraph(g,pid)[\underline{add} \mapsto add'][\underline{del} \mapsto del']$ mit
$add' = \bigcup k \in collect(pid,erzeugt,g) : \{ h(k) \}$
$del' = \bigcup k \in collect(pid,entfernt,g) : \{ h(k) \}$
wobei
h aus der Berechnung von devirtualizegraph(g,pid) zu entnehmen ist

Sofern die Devirtualisierung in Kombination mit dem Konzept zur inkrementellen Transformation mit Graphgrammatiken eingesetzt wird, ist folgende Anpassung vorzunehmen:

devirtualizeprod°(g,pid) $=_{def}$ devirtualizeprod(g,pid)[ign$\mapsto$ign'] mit
  ign' = $\bigcup$ k $\in$ collect(pid,ignoriert,g) : { h(k) }
wobei
  h aus der Berechnung von devirtualizegraph(g,pid) zu entnehmen ist

Sofern die Devirtualisierung in Kombination mit dem Konzept der Negationsbereiche eingesetzt wird, ist folgende Anpassung vorzunehmen:

devirtualizeprod°(g,pid) $=_{def}$ devirtualizeprod(g,pid)[neg$\mapsto$neg'] mit
  neg' = $\bigcup$ n $\in$ Neg$\cap$collectors(pid,in,g) mit Neg$\cap$collectors(n,umfasstNb,g)=$\emptyset$: devirtneg(n)
wobei
  Neg = instances(Negationsbereich,g),
  devirtneg(n) = [$\bigcup$ k $\in$ collect(n,umfasstGE,g) : {h(k)}] $\cup$
                 [$\bigcup$ sub $\in$ collect(n,umfasstNb,g) : devirtneg(sub)]
  h aus der Berechnung von devirtualizegraph(g,pid) zu entnehmen ist,

Die Devirtualisierung einer virtualisierten und mit pid identifizierten Produktionsregel beginnt mit der Devirtualisierung dessen Graphanteils, wodurch der Graph p entsteht. Dieser wird anschließend um die Bestandteile add und del sowie ggf. ign und neg erweitert.

Definition 2.90: Dynamische Transformationsanwendung
Für das Konzept der dynamischen Transformation ist die Definition der Transformationsanwendung wie folgt anzupassen:

trans°(g,tr) $=_{def}$ trans(g,tr $\cup$ devirtualizeprods(g))

Die für eine Transformationsanwendung einzusetzenden Produktionsregeln ergeben sich aus der Vereinigung der Transformation tr (die auch leer sein kann) und den mit der Devirtualisierungsfunktion aus g entnommenen.

2.8 Metamodelle

Mit einem Modell kann eine graphbasierte Sprache, also eine Menge von Graphen, beschrieben werden. In diesem Abschnitt werden Möglichkeiten erörtert, ein Modell selbst wieder als ein Wort einer graphbasierten Sprache zu beschreiben; oder anders formuliert, ein Modell eines Modells zu erstellen, was dann folglich als Metamodell bezeichnet wird. Auf diese Weise können Sprachdefinitionen mehrstufig und verfeinernd erfolgen.

Da ein Modell von seiner Struktur her nicht notwendigerweise dem eines Wortes gleicht (z.B. erweiternde Konzepte wie Kardinalitäten oder Constraints) werden in den folgenden Unterabschnitten die in dieser Arbeit verwendeten Möglichkeiten erörtert (vgl. Abbildung 2.58), über Modelle in beliebig vielen Meta-Ebenen zu sprechen.

	Wortsyntax	Sprachsyntax
strukturelle Ebenenorganisation Modell besitzt beliebige Struktur. Bei mehreren Ebenen werden Wort-zu-Sprache-Transformationen benötigt.	GRAPH	MODEL
virtuelle Ebenenorganisation Virtualisierung der Sprache als Wort, d.h. des Modells jeder Ebene als Graph.	GRAPH	GRAPH

Abbildung 2.58 Alternativen zur Organisation beliebig vieler Metamodellierungsebenen

Bei der in Kapitel 2.8.1 behandelten strukturellen Ebenenorganisation wird auf jeder Ebene E das entsprechende Modell m_E mit einer beliebigen Syntax definiert. Über eine Wort-zu-Sprache-Transformation wird dann aus einem zu m_E passenden Graph g_E das Modell der nächsten Ebene, m_{E+1} berechnet.

Bei der in Kapitel 2.8.2 behandelten virtuellen Ebenenorganisation wird auf jeder Ebene E das entsprechende Modell m_E mit der Struktur eines Graphs beschrieben. Um ein Modell und deren Bestandteile als solche zu erkennen, werden Typen reserviert, die von den reinen Wörtern nicht verwendet werden dürfen (siehe Virtualisierung in Kapitel 2.7.2). Anhand dieser reservierten Typen wird für die Auswertung das entsprechende Modell extrahiert. Dieses kann dabei für die Auswertung in einer beliebigen Syntax vorgelegt werden.

Der Vollständigkeit halber sei erwähnt, dass neben den vorgestellten beiden Formen noch weitere Ansätze existieren, wie z.B. eine uniforme Ebenenorganisation (vgl. [54]), bei der zwischen Wörtern und Sprachen generell nicht mehr unterschieden wird: Jedes Wort ist gleichzeitig auch eine Sprache, enthält also sowohl Anteile für „Nutzdaten“ als auch Anteile für die Beschreibung der nächsten Ebene. Im Unterschied zu virtuellen Ebenenorganisation arbeitet die Auswertung direkt auf der Syntax, d.h. es gibt keine reservierten Typen und keine Devirtualisierungen.

2.8.1 Strukturelle Ebenenorganisation

Bei der strukturellen Ebenenorganisation sind die Strukturen für Sprachdefinition und Wort voneinander verschieden, für alle Ebenen aber dieselben. Ein Anwender muss somit nur einmal und unabhängig von Ebenenorganisationen die Instanzierungssemantik verstehen, um im Rahmen einer strukturellen Ebenenorganisation arbeiten zu können. Der Ebenenwechsel erfolgt über eine Wort-zu-Sprache-Transformation. Anders als eine normale Transformation (Kapitel 2.2.4) handelt es sich hierbei um eine, die zwar einen Graph als Eingabe nimmt, doch anstelle eines Graphen eine Sprachdefinition als Ausgabe liefert.

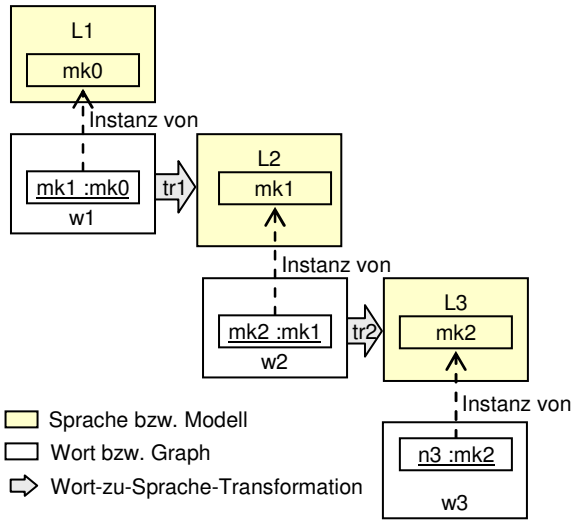


Abbildung 2.59 Strukturelle Ebenenorganisation am Beispiel

Abbildung 2.59 zeigt ein Beispiel einer strukturellen Ebenenorganisation mit drei Ebenen. Ausgehend von der zuerst gewählten Sprachdefinition L1 wird ein dazu passendes Wort w1 erstellt. Durch Anwendung der Wort-zu-Sprache-Transformation tr1 wird w1 in die Sprachdefinition L2 umgewandelt, und damit die nächste Ebene betreten. Die Umwandlung basiert in diesem Beispiel darauf, dass jeder Knoten eines Wortes zu einem neuen Modellknoten der erzeugten Sprachdefinition umgewandelt wird, wobei die Id zum Typ wird. Der nächste Ebenenwechsel (über die Transformation tr2) erfolgt analog.

Als konkrete Technik für Wort-zu-Sprache-Transformationen wird in dieser Arbeit die Kombination aus einer einfachen Transformation, die aus einem Wort (bzw. Graph) eine als Wort virtualisierte Sprache (bzw. Modell) erzeugt, und der Devirtualisierungsfunktion (von Graphen zu Modellen) verwendet. Als Grundlage dazu wird die Virtualisierung von Graphen aus Kapitel 2.7.2 herangezogen und auf Modelle erweitert. Abbildung 2.60 stellt das resultierende Virtualisierungsmodell für Modell dar, wobei die Erweiterungen gegenüber dem Virtualisierungmodell für Graphen grün eingefärbt sind.

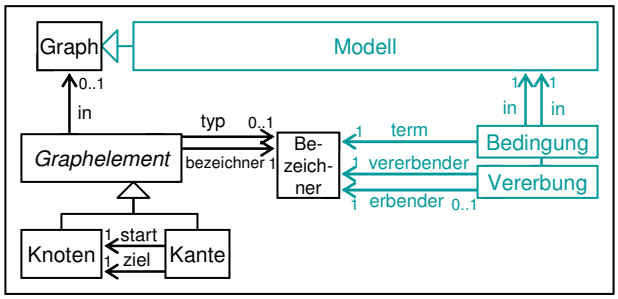


Abbildung 2.60 Virtualisierungsmodell für Modelle

Abbildung 2.61 zeigt (auf der linken Seite) Produktionsregeln mit denen die Transformation tr1 aus Abbildung 2.59 definiert werden kann. Mit p1 wird zunächst der „Hauptknoten“ des virtualisierten Modells erstellt (ein Graphelement des Typs Modell). An dieses werden mit p2 die Konstrukte des Modells – ebenfalls in virtualisierter Form – angehängt. Dabei wird jedes Graphelement mit Typ mk0 aus w1 (welches in Abbildung 2.59 die Eingabe für tr1 ist) zu einem virtualisierten Konstrukt (der so entstehenden Sprache L2) übersetzt. Auf der rechten Seite der Abbildung 2.61 wird eine verkürzte Notation gezeigt, die sich an der Notation für virtualisierte Produktionsregeln orientiert: Dem „Hauptknoten“ untergeordnete Bestandteile werden innerhalb eines darunter angebrachten Rahmens mit gepunkteter Linie dargestellt. Da es sich bei p1 und p2 nicht nur um Graphen, sondern um Produktionsregeln handelt, wird

noch die Darstellung der neu erzeugten Elemente benötigt. Dazu wird das Farbschema von Produktionsregeln übernommen: Grün für neu erzeugte Elemente. Allerdings werden neu erzeugte virtualisierte Graphelemente nicht mit grüner Linienfarbe dargestellt, sondern mit einem grünen, umgebenden Bereich. Das lässt die Option offen, nicht nur Modelle zu erzeugen, sondern auch Produktionsregeln.

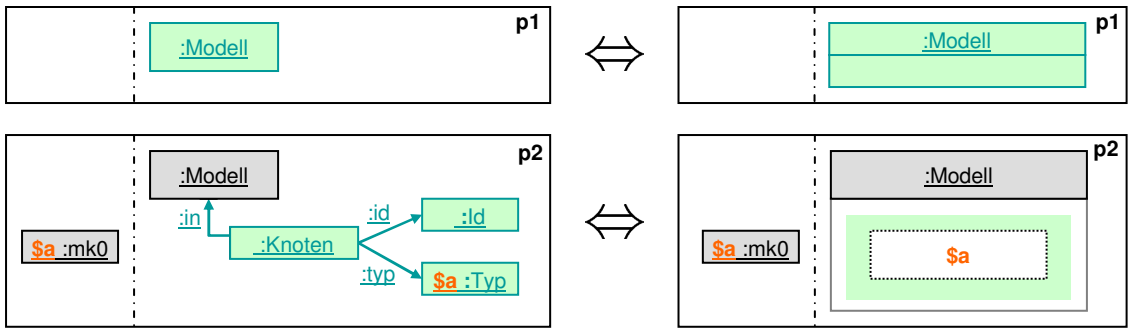


Abbildung 2.61 Notation für virtualisierte Modellbestandteile

Definition 2.91: Devirtualisierungsfunktion für Modelle
Die ***Devirtualisierungsfunktion*** devirtualizemods *nimmt eine Instanz des Virtualisierungsmodells, für Modelle als Eingabe, und extrahiert daraus die virtualisierten Modelle als „echte" Elemente von* MODEL*, und ist wie folgt definiert:*

$$\text{devirtualizemods}(g) =_{def} \bigcup \text{mid} \in \text{instances}(\text{Modell},g) : \{ \text{devirtualizemod}(g, \text{mid}) \}$$

Die Funktion devirtualizemod *extrahiert eine einzelne Produktionsregel, die mit der als Parameter angegebenen Id* mid*, aus einer Instanz des Virtualisierungsmodells* g*, und ist wie folgt definiert:*

$$\text{devirtualizemod}(g,\text{mid}) =_{def} \text{devirtualizegraph}(g,\text{mid})$$

Sofern die Devirtualisierung in Kombination mit dem Konzept der Constraints angewendet wird, ist folgende Anpassung vorzunehmen:

$$\text{devirtualizemod}'(g,\text{mid}) =_{def} \text{devirtualizemod}(g,\text{mid})[\underline{\text{Cnst}} \mapsto C'] \text{ mit}$$
$$C' = \bigcup b \in \text{instances}(\text{Bedingung},g) \cap \text{collectors}(\text{mid},\text{in},g) : \text{collect}(b,\text{term},g)$$

Sofern die Devirtualisierung in Kombination mit dem Konzept der Vererbung angewendet wird, ist folgende Anpassung vorzunehmen:

$$\text{devirtualizemod}'(g,\text{mid}) =_{def} p[\underline{\text{inherit}} \mapsto \text{inherit}'] \text{ mit}$$
$$\text{inherit}' = \bigcup v \in \text{instances}(\text{Vererbung},g) \cap \text{collectors}(\text{mid},\text{in},g) :$$
$$\{(\text{any}(\text{collect}(v,\text{erbender},g)) , \text{any}(\text{collect}(v,\text{vererbender},g)))\}$$

Die Devirtualisierung eines virtualisierten und mit mid identifizierten Modells beginnt mit der Devirtualisierung dessen Graphanteils, wodurch der Graph m entsteht. Dieser wird anschließend um die Bestandteile Cnst und inherit erweitert.

2.8.2 Virtuelle Ebenenorganisation

Bei der virtuellen Ebenenorganisation wird die Sprachdefinition jeder Ebene in der Struktur eines Wortes abgelegt. Ein Modell ist folglich nicht mehr explizit, sondern nur virtuell, als ein Wort kodiert, verfügbar. Um ein so virtualisiertes Modell von dem eigentlichen Wort zu unterscheiden (für den die Vorgaben gelten), werden reservierte Typen verwendet. Dazu wird

eine Sprache definiert, die diese reservierten Typen enthält. Abbildung 2.62 zeigt dies anhand eines Beispiels: Die Ausgangssituation (links im Bild) besteht aus einem Wort w3 dass zu einem Modell L2 passt. Um L2 zu einem Wort w2 zu virtualisieren, wird zunächst durch die Sprache L festgelegt, wie Modelle als Wörter abzulegen sind. Hier ist jeder „Modellknoten" als ein Knoten vom Typ Modellknoten abzulegen, wobei die Id als Informationsträger[18] zu verwenden ist. Der Typ mk2 (links) wird daher als die Id mk2 (rechts) geführt – in einem Knoten vom Typ Modellknoten. Da die Typen nun flexibel sind, und damit nicht mehr in L vorkommen, kann die Typbindung der Graphelemente eines Wortes nicht mehr explizit wie in der Ausgangssituation erfolgen. Daher wird nicht nur ein Modell, sondern auch ein Wort selbst virtualisiert. Der Knoten n3 mit Typ mk2 (aus w3) wird in der Virtualisierung durch den Knoten n3 vom Typ Knoten dargestellt. Die Typbindung zu mk2 erfolgt über eine Kante vom Typ typ zum Knoten mit Id mk2 und Typ Modellknoten in w2. Da somit Kanten über die Wortgrenzen verlaufen, kann das zu w3' virtualisierte w3 nur im Zusammenhang mit w2 existieren. Alternativ können beide auch zu einem Wort zusammengefasst werden. Vom Vorgehen her würde dann zunächst w2 als „Modellebene" erstellt, und dann um Inhalte der „Wortebene" erweitert werden.

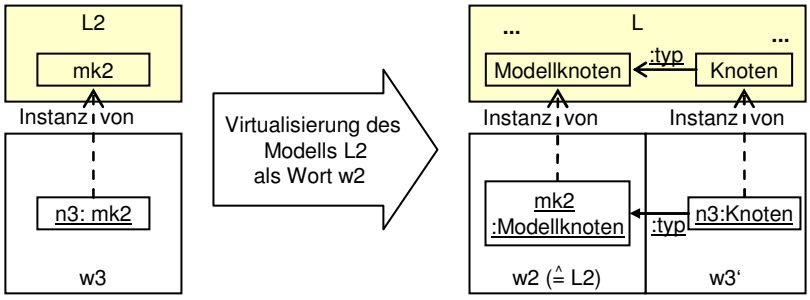


Abbildung 2.62 Virtualisierung eines Modells als Wort

Obwohl L hier selbst wieder ein Modell ist, enthält L keinerlei „fachliche" Informationen, sondern dient ausschließlich als „technischer" Rahmen. Die Prüfung, ob ein Wort w3 Instanz eines Modells L2 ist, umspannt damit sowohl die „technische" Prüfung (d.h. w3 muss in der Extension von L sein) als auch die „fachliche" Prüfung hinsichtlich des virtualisierten Modells. Für letztere wird eine durch eine Extraktionsfunktion aus dem virtualisierten Modell ein „normales" Modell herausgezogen, und eine „technische" Prüfung durchgeführt. Dieser Schritt entspricht der Wort-zu-Sprache-Transformation bei einer strukturellen Ebenenorganisation. Der Unterschied besteht jedoch darin, dass eine Person, die ein Wort w3 gemäß L2 erstellt, L2 nicht wie bei einer strukturellen Ebenenorganisation als „normales" Modell, sondern als ein virtuelles (d.h. in Form eines Wortes) vorliegen hat.

Um beliebig viele Ebenen zu ermöglichen, kann die Systematik auf gleiche Weise fortgesetzt werden. Abbildung 2.63 (links) zeigt das um eine weitere Modellebene erweiterte Beispiel aus Abbildung 2.62. Konkret ist dabei der Knoten mk2 eine Instanz des Knotens mk1 des mit w1 virtualisierten Modells. Da w2 selbst ein Modell ist, beschreibt w1 damit ein Modell eines Modells, und wird daher als Metamodell bezeichnet. Um die Elemente des Metamodells vom Modell zu unterscheiden, wird in L der Typ Metamodellknoten eingeführt. Obwohl dies für weitere Ebenen beliebig so fortgesetzt werden kann (für die nächste Ebene wäre der Typ Metametamodellknoten nötig), wäre die Anzahl der Ebenen stets vorherbestimmt. Um auch davon zu abstrahieren, können auch die Ebenen als Bestandteil der Wörter virtualisiert werden. Abbildung 2.63 (rechts) zeigt die sich aus dieser Idee ergebende Struktur, bei der

[18] Der Typ eines Knotens steht als Informationsträger nicht mehr zur Verfügung da er für die Unterscheidung zwischen virtualisierten Modellelementen und den eigentlichen Wortelementen verwendet wird. Alternativ zur Id könnten auch Attribute bzw. Slots verwendet werden.

einerseits L fix ist, andererseits beliebig viele Ebenen mit beliebigen Modellen bzw. Wörtern wählbar sind.

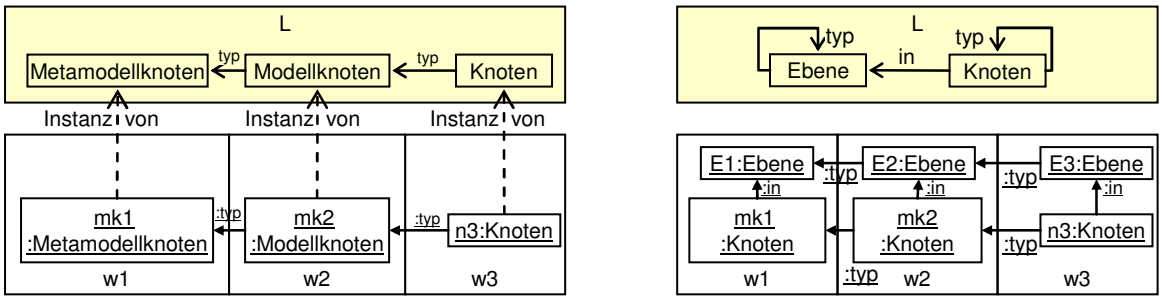


Abbildung 2.63 Virtuelle Ebenenorganisation am Beispiel

Für die praktische Verwendung kann es hilfreich sein, die oberste Ebene, sofern diese fixiert werden kann, in die Sprache L wieder zu ent-virtualisieren – wie in Abbildung 2.64 (links) gezeigt. Damit können aus der obersten Ebene Vorgaben an die Typen sämtlicher Ebenen direkt gestellt[19] werden (Abbildung 2.64 rechts).

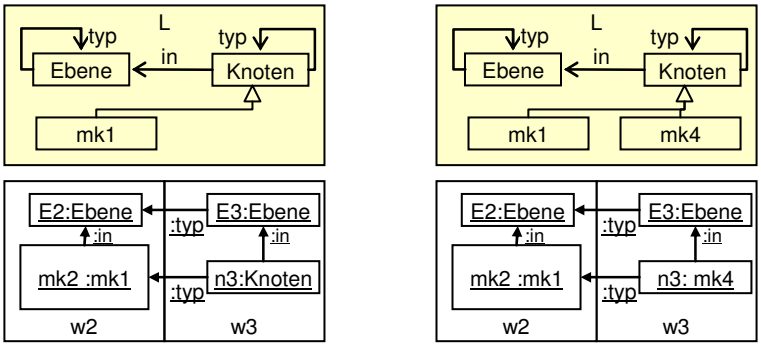


Abbildung 2.64 Entvirtualisierung der obersten Ebene für globale Typvorgaben

[19] Um die verwendbaren Typen auf die jeweiligen Ebenen zu beschränken, müssen noch entsprechende Constraints hinzugefügt werden.

2.9 Zusammenfassung

Abbildung 2.65 stellt die in diesem Kapitel definierten Strukturen und diesbezüglich erweiternde Konzepte im Überblick dar. Als Notation wird eine Modifikation eines UML-Klassendiagramms verwendet:

- Jede Klasse steht für eine Mengendefinition, jedes Objekt einer Klasse entspricht einem Element der definierten Menge.
- Erbt eine Klasse k1 von einer Klasse k2, so ist die durch k1 definierte Menge eine Teilmenge von der durch k2 definierten Menge.
- Hat eine Klasse Attribute der Form „x y“ mit $x \in PSYMBOL$, so ist jedes Objekt dieser Klasse eine Struktur (siehe 2.1.2). Jedes „x y“ beschreibt, dass jenes Objekt den Bestandteil x hat welches wiederum ein y ist.
- Hat eine Klasse Attribute der Form „π_i y“ mit $i \in \mathbb{N}$, so ist jedes Objekt dieser Klasse ein n-Tupel, wobei n der Anzahl der Attribute entspricht. Jedes „π_i y“ beschreibt, dass die Tupelposition i ein y ist.
- Hat eine Klasse genau ein Attribut der Form „y“, die nicht auch den vorangegangen Punkten entspricht, so ist jedes Objekt dieser Klasse genau ein y.
- Hat eine Klasse keine Attribute, so ist über ein Objekt dieser Klasse nichts bekannt – es kann ein beliebiges mathematisches Objekt sein.
- Jede Farbe steht für ein Erweiterungskonzept. Mit entsprechender Farbe unterlegte Elemente des Klassendiagramms kommen bei Anwendung jenes Konzepts hinzu – bzw. sind nicht vorhanden, wenn das Konzept nicht angewendet wird.

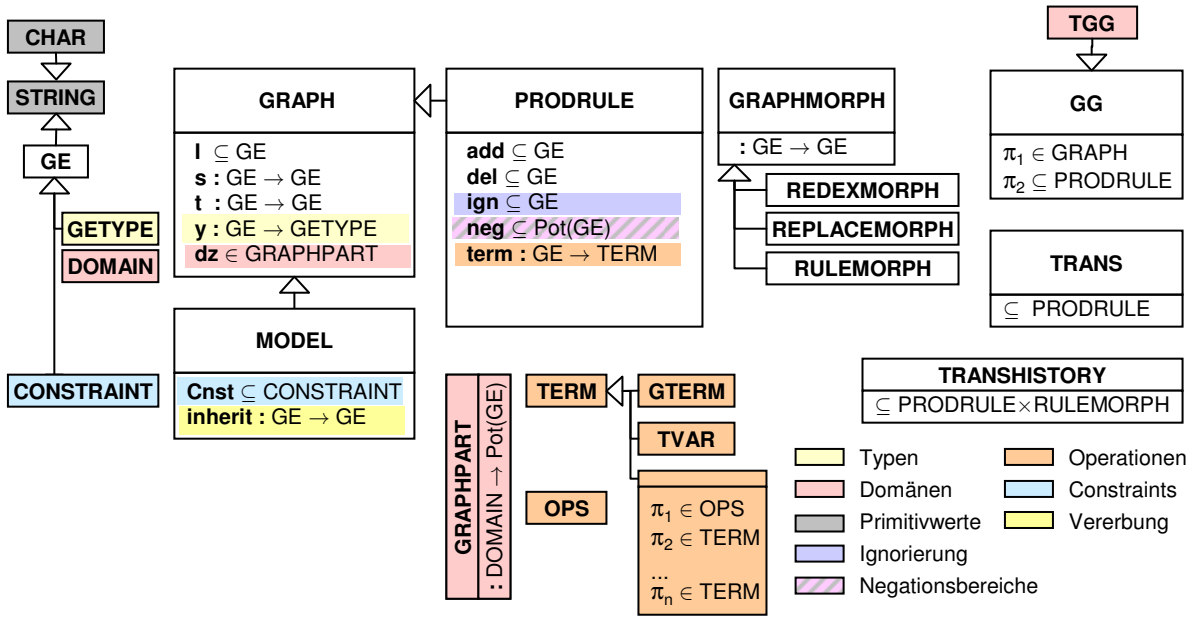


Abbildung 2.65 Überblick über definierte Strukturen und erweiternde Konzepte

Abbildung 2.66 stellt die definierten Berechnungsvorschriften für Transformationen hinsichtlich der verwendeten Konzepte gegenüber. Schwarz gefärbte Anteile sind bei allen Transformation-Berechnungsvorschriften in identischer Form vorhanden. Ocker gefärbte Anteile kommen beim Einsatz der Operationen (Kapitel 2.4.7) hinzu, grau gefärbte Anteile beim Einsatz der inkrementellen Transformation (Kapitel 2.3.1), dunkelrot gefärbte beim Einsatz von redexunikaten (Kapitel 2.4.2), rosa gefärbte Anteile beim Einsatz von Domänen (siehe Kapitel 2.4.2), blau gefärbte Anteile bei Einsatz der Ignorierung (Kapitel 2.4.3) und hellgrüne Anteile beim Einsatz dynamsicher Transformationen (Kapitel 2.7).

Bei einer redexunikaten und inkrementellen Transformation enthalten die Parameter c1 und cu die gleichen Werte und werden im Rahmen der Rekursion auch auf gleiche Weise fortgeschrieben. Daher können diese zu einem Parameter zusammengefasst werden. In der Darstellung in Abbildung 2.66 wurde darauf bewusst verzichtet, um die einzelnen Konzepte besser hervorzuheben.

$inktrans_1(g,tr,c0,a,b) =_{def}$ resolveid($inktrans_1$(g[dom ↦ {a↦Id_g}] ,tr ∪ devirtualizeprods(g),c0,∅,∅) $|_G$ $dom_g(b)$)

$inktrans_1(g,tr,c0,c1,cu) =_{def}$ falls |K|=0 dann (g,c1) sonst $inktrans_1$(g'',tr,c0,c1'',cu'')

wobei

K = [∪ p∈tr : {p}×[∪ k∈pmf(p,g) : { k ↾ (Id_p/ign_p) }]] / Q

Q = ∪ (p,hr)∈c1 : { (p,redexmorph(hr,p)) }

(p,hx) = rsf(K)

[falls rulemorph(c0,p,hx) def. dann

g'' = rereduce(g,p,rulemorph(c0,p,hx))

hc = replacemorph(rulemorph(c0,p,hx),p)

sonst

(g'',hc) = reduce(g,p,hx), einmalig ausgeführt]

c1'' = c1 ∪ {(p,hx ∪ hc)}

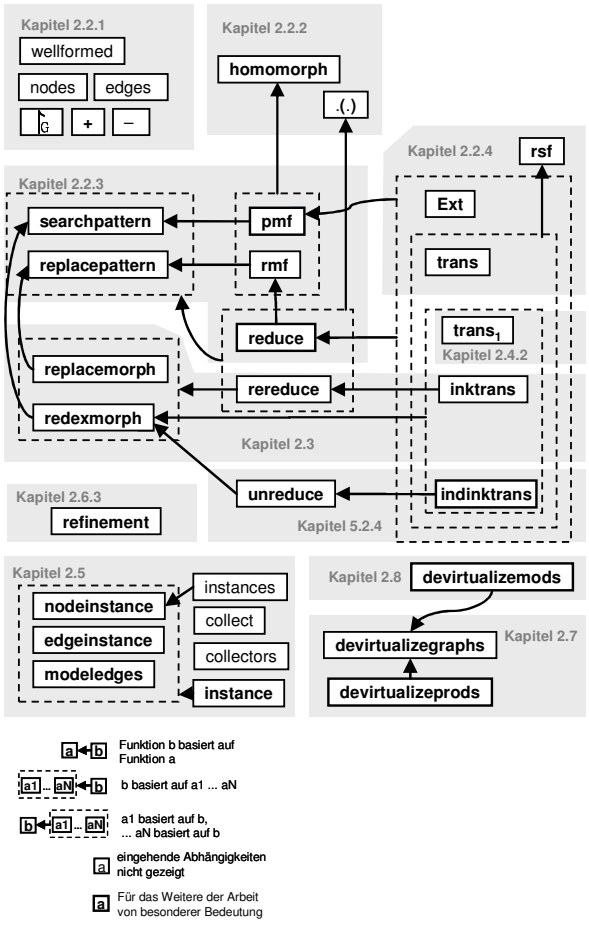
cu'' = cu ∪ {(p,hx ∪ hc)}

Transformation

+ inkrementell
+ redexunikat
+ Domänen
+ Ignorierung
+ Operationen
+ dynamisch

Abbildung 2.66 Gegenüberstellung der Transformation-Berechnungsvorschriften

Abbildung 2.67 zeigt die wesentlichsten, in diesem Kapitel definierten Funktionen im Zusammenhang und mit ihrer Signatur dar. Funktionsanpassungen werden der Übersichtlichkeit halber nicht dargestellt.



Kapitel 2.2.1 Graphen als Struktur für Wörter

$|_G$: GRAPH x Pot(GE) -> GRAPH
+ : GRAPH x GRAPH -> GRAPH
- : GRAPH x GRAPH -> GRAPH
wellformed : GRAPH -> BOOL
nodes : GRAPH -> Pot(GE)
edges : GRAPH -> Pot(GE)

Kapitel 2.2.2 Graphhomomorphismus

homomorph: GRAPHMORPH x GRAPH x GRAPH -> BOOL
.(.): GRAPHMORPH x GRAPH -> GRAPH

Kapitel 2.2.3 Produktionsregel

searchpattern: PRODRULE -> PRODRULE
replacepattern: PRODRULE -> PRODRULE
pmf: GRAPH x PRODRULE -> Pot(REDEXMORPH)
rmf: GRAPH x PRODRULE x REDEXMORPH -> REPLACEMORPH
reduce: GRAPH x PRODRULE x REDEXMORPH -> REPLACEMORPH

Kapitel 2.2.4 Graphgrammatik und Transformation

Ext: GG -> Pot(GRAPH)
rsf: Pot(PRODRULE x REDEXMORPH) -> PRODRULE x REDEXMORPH
trans: GRAPH x TRANS -> GRAPH

Kapitel 2.3 Aktualisierung einer Transformationsanwendung

replacemorph: RULEMORPH x PRODRULE -> REPLACEMORPH
redexmorph: RULEMORPH x PRODRULE -> REDEXMORPH
rereduce: GRAPH x PRODRULE x REDEXMORPH -> REPLACEMORPH
inktrans: GRAPH x TRANS x TRANSHISTORY-> GRAPH x TRANSHISTORY

Kapitel 2.4.2 Transformation mit Graphgrammatiken

$trans_1$: GRAPH x TRANS -> GRAPH

Kapitel 2.5 Modelle

nodeinstance: GE x GRAPH x GE x MODEL -> BOOL
edgeinstance: GE x GRAPH x GE x MODEL -> BOOL
modeledges: GE x MODEL -> Pot(GE)
instance: GRAPH x MODEL -> BOOL
instances: GETYPE x GRAPH [x MODEL] -> Pot(GE)
collect: GE x GETYPE x GRAPH [x MODEL] -> Pot(GE)
collectors: GE x GETYPE x GRAPH [x MODEL] -> Pot(GE)

Kapitel 2.6.3 Vererbung

refinement: MODEL x MODEL -> BOOL

Kapitel 2.7 Dynamische Transformation

devirtualizegraphs: GRAPH -> Pot(GRAPH)
devirtualizeprods: GRAPH -> Pot(PRODRULE)

Kapitel 2.8 Metamodelle

devirtualizemods: GRAPH -> Pot(MODEL)

Abbildung 2.67 Überblick zu der Verwendung der Definitionen

3 Grundlagen zum Projektcontrolling

In diesem Kapitel werden die Begriffe eingeführt, die als Grundlage zur Definition des Projektcontrollings dienen und somit die Begriffe des Projekts, des Vorgehensmodells und der Prozessbeschreibungssprache umfassen. Soweit für das Weitere der Arbeit gebraucht, werden konkrete Beispiele aus der Praxis vorgestellt.

3.1 Projekt, Vorgehensmodell, Prozessbeschreibungssprache

Projekt. Für den Begriff des Projektes sei auf die DIN 69901 [55] zurückgegriffen, nach der ein Projekt *„ein Vorhaben [ist], das im Wesentlichen durch eine Einmaligkeit der Bedingungen in ihrer Gesamtheit gekennzeichnet ist. Ein Projekt ist definiert durch die Zielvorgabe, die zeitlichen, finanziellen und personellen Rahmenbedingungen, sowie allgemein durch die Abgrenzung gegenüber anderen Vorhaben.“*

Vorgehensmodell. Ein Vorgehensmodell gibt vor *was*, *wie, durch wen* und *wann* in einem Projekt zu erstellen, abzuschließen oder zu tun ist. Darüber hinaus kann ein Vorgehensmodell auch Vorgaben zu seiner eigenen *Anpassung* enthalten, die im Weiteren als Steuerungsvorgaben bezeichnet werden. Die Vorgaben eines Vorgehensmodells machen implizit Annahmen über die Elemente eines Projektes sowie deren Struktur. Beispielsweise basiert das *Was* auf Dokumenten oder anderweitigen Ergebnissen die in einem Projekt vorkommen, denen das Vorgehensmodell Typen, Beschreibungen oder Methoden zuordnet. Im Bezug auf die Typung fungiert ein Vorgehensmodell damit als eine Sprache (bzw. Modell im Sinne von Kapitel 2.5), und sieht das Projekt entsprechend als Wort (bzw. Graph) an.

Prozessbeschreibungssprache. Es gibt nicht DAS Vorgehensmodell, dessen Vorgaben für alle Projekte gleichermaßen hilfreich sind. Beispielsweise benötigen große Projekte mehr Organisation als kleine, Projekte zur Systementwicklung laufen anders ab als solche zur Beschaffung. Folglich kann die Menge der Projekte bezüglich der Gemeinsamkeiten und Unterschiede klassifiziert werden, um je ein darauf abgestimmtes Vorgehensmodell anzuwenden. Vorgehensmodelle werden damit selbst Gegenstand einer Bearbeitung, die auf Organisationsebene (durch den Prozessingenieur) oder sogar während eines Projektes (durch den Projektleiter) erfolgen kann. Um auch dieser Bearbeitung einen Rahmen zu geben, werden Prozessbeschreibungssprachen verwendet, die Vorgehensmodelle als Wörter betrachten.

Es gibt auch nicht DIE Prozessbeschreibungssprache, mit der sich alle Vorgehensmodelle gleichermaßen gut beschreiben lassen. Neben Unterschieden in der konkreten Syntax liegen denen auch verschiedene Konzepte zu Grunde. Die Definition einer Prozessbeschreibungssprache erfolgt häufig über deren abstrakte Syntax, die wiederum auf analytische Weise (vgl. [56]) beschrieben ist. Eine Prozessbeschreibungssprache ist also ein Modell. Als konkrete Syntax wird oft ein Derivat von XML [57] definiert, mit dem ein Prozessingenieur üblicherweise aber nicht direkt zu tun hat. Stattdessen werden meist graphische Werkzeuge eingesetzt, die sich an der abstrakten Syntax orientieren. Entsprechend wird in dieser Arbeit auf die Betrachtung der konkreten Syntax von Prozessbeschreibungssprachen weitestgehend verzichtet.

Ein wesentlicher Unterschied zwischen Prozessbeschreibungssprachen und Programmiersprachen (wie z.B. Java) besteht darin, dass erstere auf die Definition von Vorgehensmodellen „zugeschnitten“ sind. Abbildung 3.1 veranschaulicht den Unterschied an einem Beispiel: Während eine Prozessbeschreibungssprache für Bestandteile von Vorgehensmodellen direkte Konstrukte (wie Rolle) anbietet, müssen diese bei Anwendung einer Programmiersprache vom Anwender erst definiert werden. Auch wenn jene Anteile z.B. in Form einer

Programmbibliothek bereitgestellt wären, blieben die Typ-Beziehungen weiterhin virtualisiert, was deren Anwendung aufwendiger gestaltet als eine Nutzung direkter Konstrukte (wie rechts in Abbildung 3.1).

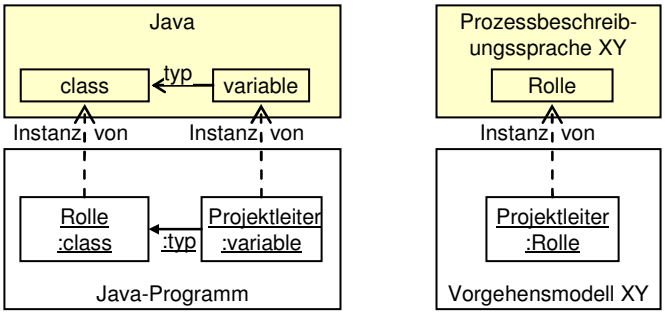


Abbildung 3.1 Abstraktionsniveaus bei Programmier- und Prozessbeschreibungssprachen

Zusammenhang zwischen Projekt, Vorgehensmodell und Prozessbeschreibungssprache. Entsprechend der beiden vorangegangen Paragraphen fungiert ein Vorgehensmodell sowohl als Sprache (für Projekte) als auch als Wort (einer Prozessbeschreibungssprache). Die sich daraus ergebende Frage ist, wie dies in Einklang zu bringen ist. Diesbezüglich grundlegende Überlegungen wurden im Kapitel 2.8 ausgeführt. Darauf basierend wird für diese Arbeit die virtuelle Ebenenorganisation gewählt, wodurch Projekte und Vorgehensmodelle durch Graphen, und Prozessbeschreibungssprachen durch Modelle dargestellt werden (vgl. Abbildung 3.2).

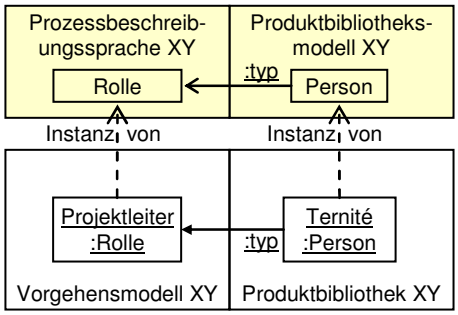


Abbildung 3.2 Zusammenhang Projekt, Vorgehensmodell, Prozessbeschreibungssprache

Definition 3.1: Projekt
Die Menge PRJ *aller* ***Projekte*** *ist wie folgt definiert:*

$$PRJ = GRAPH$$

Definition 3.2: Vorgehensmodell
Die Menge VM *aller* ***Vorgehensmodelle*** *ist wie folgt definiert:*

$$VM = GRAPH$$

Definition 3.3: Prozessbeschreibungssprache
Die Menge PBS *aller* ***Prozessbeschreibungssprachen*** *ist wie folgt definiert:*

$$PBS = MODEL$$

3.2 Beispiele für Prozessbeschreibungssprachen

In diesem Kapitel werden aktuelle Prozessbeschreibungssprachen vorgestellt. Gleichzeitig wird dabei aufgezeigt, dass es mit diesen Prozessbeschreibungssprache nicht möglich ist, Vorgaben so zu definieren, dass sie im Sinne einer inkrementellen Transformation auf ein Projekt angewendet werden können. Der Umgang mit Änderungen muss also von einem Prozessingenieur selbst ausformuliert werden. Die Prozessbeschreibungssprache nimmt ihm das nicht ab[20]. Bei Einsatz von Graphgrammatiken müsste sich ein Entwickler zwar nicht um den Umgang mit Änderungen kümmern (vgl. Kapitel 2.3 bzw. 2.4.3), allerdings sind Graphgrammatiken für die Praxis auf einem zu niedrigem Abstraktionsniveau. Die Auswahl der vorgestellten Prozessbeschreibungssprachen richtet sich hauptsächlich nach deren Verwendung in der Praxis – als Beleg eines geeigneten Abstraktionsniveaus.

Konkret werden die Prozessbeschreibungssprache JIL (in Kapitel 3.2.1), XPDL 2.1 (in Kapitel 3.2.2), SPEM 2.0 (in Kapitel 3.2.3) und V-Modell XT 1.3 Metamodell (in Kapitel 3.2.4) behandelt. Diese vier Prozessbeschreibungssprachen sind im Hinblick auf Abbildung 3.1 nicht als gleichwertig einzustufen. Vielmehr decken sie in der aufgeführten Reihenfolge das Spektrum zwischen Programmier- und Prozessbeschreibungssprachen ab. So ist JIL eher als eine Programmiersprache anzusehen, die um Konstrukte für Vorgehensmodelle angereichert wurde: Von den vier Prozessbeschreibungssprachen enthält sie noch den meisten Konstrukte einer typischen Programmiersprache. XPDL dagegen konzentriert sich auf den Kontroll- und Datenfluss. Berechnungsdetails werden auf externe Quellen ausgelagert. SPEM setzt diesen Weg fort und lagert die Details zum Ablaufverhalten weitestgehend aus. In der letzten Stufe, mit dem V-Modell XT 1.3 Metamodell, wird der Ablauf nur noch in einer ganz groben Form verfolgt. Stattdessen liegen die Projektergebnisse und deren Erzeugungsabhängigkeiten im Vordergrund. Diese werden zudem generell unabhängig von einem bestimmten Ablauf definiert.

Die Prozessbeschreibungssprachen werden jeweils nur hinsichtlich eines solchen Ausschnitts vorgestellt, der zur Klärung der Fähigkeit zum Umgang mit Änderungen (im Sinne einer inkrementellen Transformation) relevant ist. Um die Gemeinsamkeiten und Unterschiede deutlich herauszustellen, werden die Konstrukte der Prozessbeschreibungssprachen umbenannt und nur sinngemäß wiedergegeben. Gleichnamige Konstrukte zweier Prozessbeschreibungssprachen stellen dabei ähnliche, allerdings nicht notwendigerweise in jedem Detail exakt übereinstimmende Konzepte dar (siehe dazu auch [58]).

Um den Unterschied der Prozessbeschreibungssprachen zu Programmiersprachen darzustellen, definiert Abbildung 3.3 eine „typische“ Programmiersprache. So erlaubt es deren Syntax Programme sowohl funktional (vgl. [34]) als auch imperativ (vgl. [59]) zu beschreiben. Der Umfang der Syntax ist so gewählt, dass sie die wesentlichen Konzepte zur Diskussion der Prozessbeschreibungssprachen enthält[21]. Sofern ein Funktionsrumpf, oder der „Rumpf“ einer Kontrollstruktur wie Fallunterscheidung oder Schleife, über mehrere Anweisungen verfügt, wird implizit die Existenz einer darüber definierten Ordnung angenommen. Notiert wird dies – wie bei UML-Klassendiagrammen – durch den Zusatz „{ordered}“ an der entsprechenden Assoziation[22]. Bei Funktionen werden anstelle eines Rückgabewerts Formalparameter mit einem Modus ausgestattet. Dieser Modus erlaubt es, bei

[20] Zum Vergleich: In der Programmiersprache Java wird einem Entwickler die Arbeit abgenommen, sich um die Freigabe reservierten Speichers zu kümmern.

[21] Weshalb andere Konzepte, wie die logische Programmierung (vgl. [60]) oder Objektorientierung (vgl. [61]), hier nicht mit erfasst sind.

[22] Gemäß Kapitel 2 sind alle Kanten eines Graphen als Menge abgelegt. Eine Ordnung über dessen Kanten kann damit nur virtuell durch weitere Knoten und Kanten ausgedrückt werden – für die es auf Modellebene entsprechender Modellelemente bedarf. Zur Übersichtlichkeit wird auf deren explizite Angabe verzichtet.

einem Funktionsaufruf eingesetzte Aktualparameter aus der ausgerufenen Funktion heraus zu verändern – und damit eine Rückgabe durchzuführen. Dieses Verhalten entspricht beispielsweise den Referenzparametern in C++. Die Bindung von Aktual- zu Formalparametern wird anhand deren Reihenfolge vorgenommen. Entsprechend wird dabei im Modell ebenfalls implizit die Existenz einer definierten Ordnung angenommen. Ein Funktionsabbruch beendet die aktuell ausgeführte Funktion (vgl. *return*-Konstrukt in Java), ein Schleifenabbruch die aktuell ausgeführte Schleife (vgl. *break*-Konstrukt).

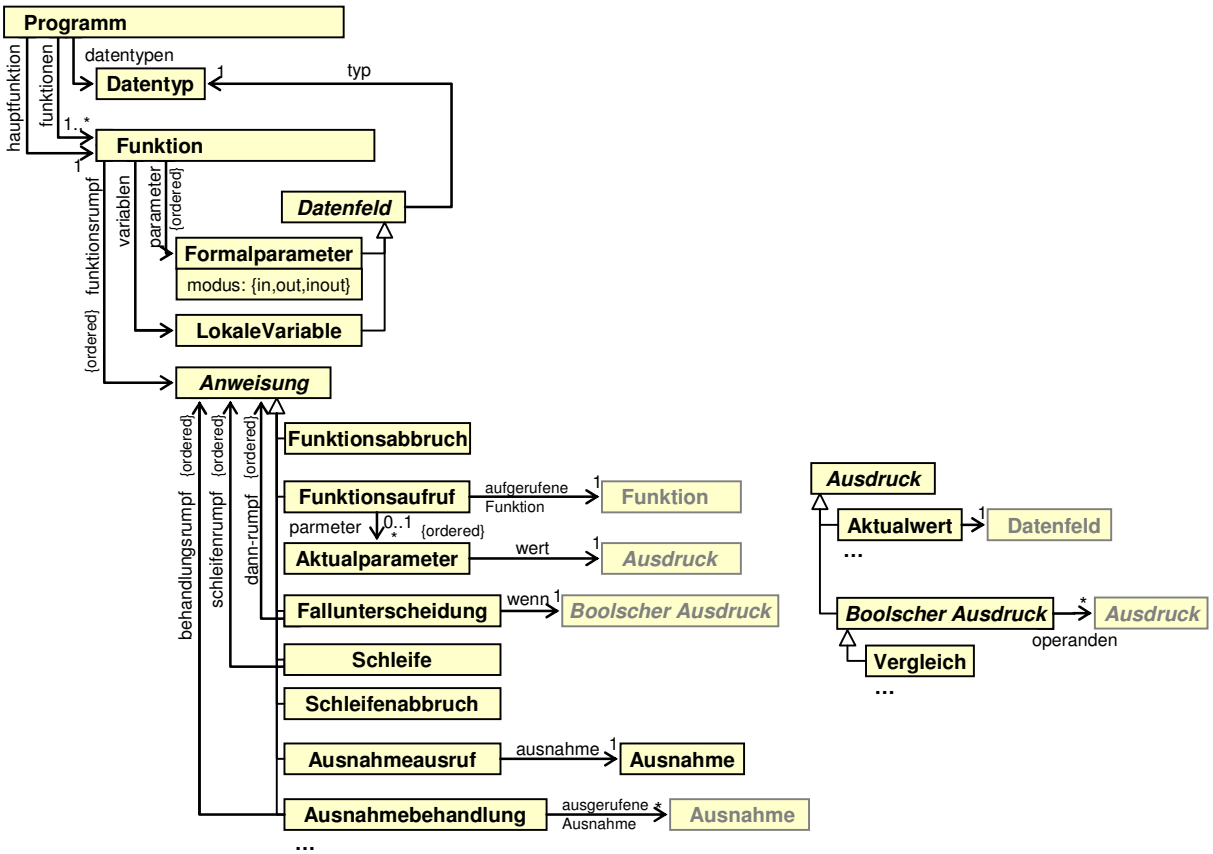


Abbildung 3.3 Typische Programmiersprache, abstrakte Syntax, sinngemäß

3.2.1 JIL

Die Prozessbeschreibungssprache JIL [63,64] vereinigt die Konzepte einer Reihe von Sprachen zu denen u.a. APPL/A [65], FUNSOFT [66], EPOS [67], Marvel [68,69], Merlin [70,71] und Process Weaver [72] gehören. Dadurch stellt JIL als Sprache eine interessante „Zusammenfassung“ dar, auch wenn sie sich derzeit noch in Entwicklung befindet und keinen praktischen Einsatz zur Beschreibung von Vorgehensmodellen erfahren hat.

Abbildung 3.4 zeigt die sinngemäße, abstrakte Syntax von JIL. Gelb eingefärbt sind Konstrukte, die so auch in der „typischen“ Programmiersprache zu finden sind. Die für Prozessbeschreibungssprachen spezifischen Konstrukte werden im Folgenden vorgestellt.

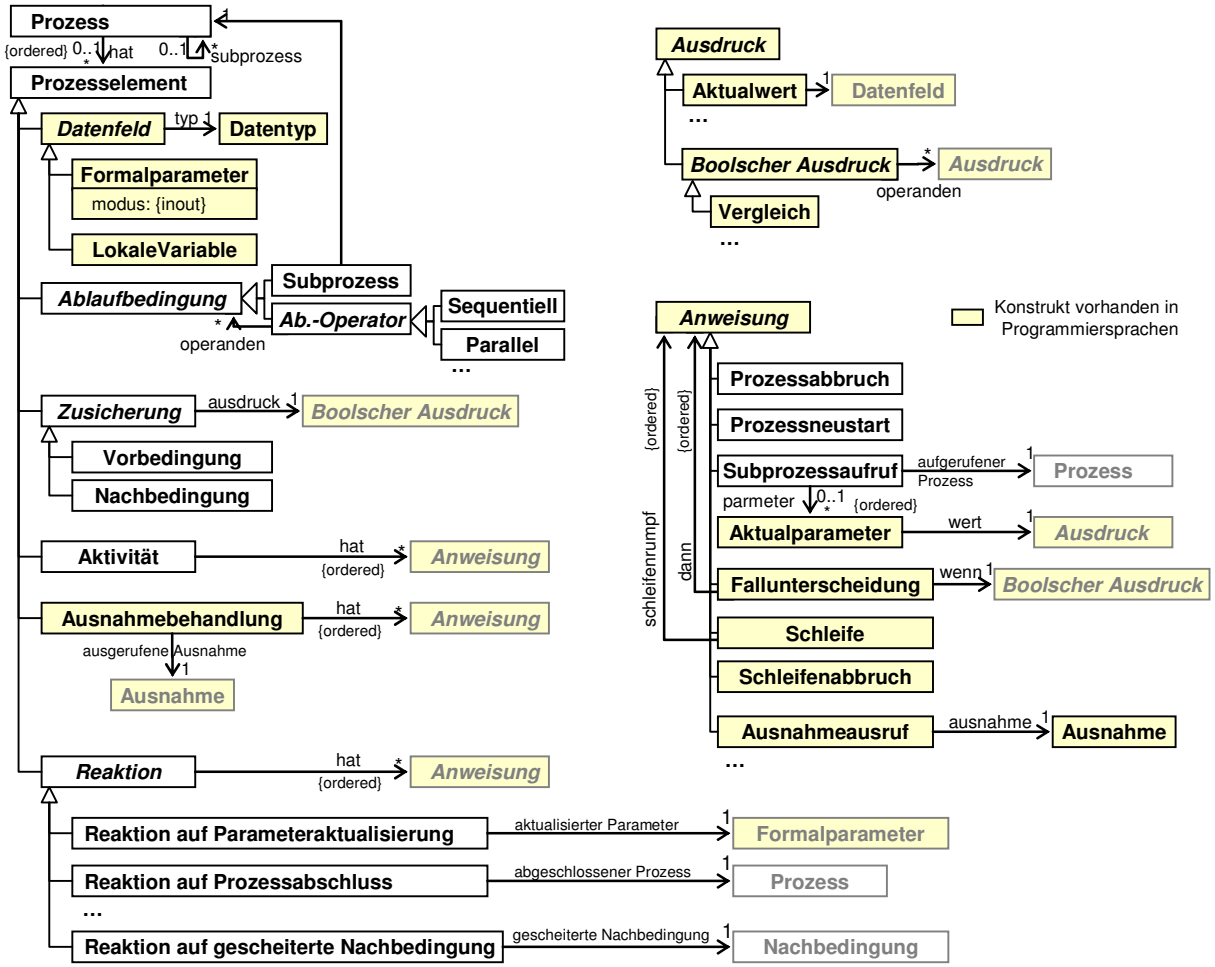


Abbildung 3.4 JIL, abstrakte Syntax, sinngemäß

Prozess und Aktivität. Zur Beschreibung von Vorgehensmodellen steht ein Prozess für den Ablauf des gesamten Projekts oder einen Teilablauf darin. Ein Prozess gliedert sich, neben Teilprozessen, in Aktivitäten. Eine Aktivität steht für eine Aufgabe die einer Person (oder einer maschinellen Ressource) zur Erledigung zugewiesen werden kann.

Strukturell ist ein Prozess mit einer Funktion bei Programmiersprachen vergleichbar: So besitzt dieser ebenfalls Formalparameter und lokale Variablen, für die beim Aufruf des Prozesses pro und für die Dauer des Aufrufs ein Speicherplatz reserviert wird. Die Aktivitäten des Prozesses entsprechen dem Funktionsrumpf, wobei diese bei JIL sich wiederum aus Anweisungen zusammensetzen können. Eine Vielzahl davon ist aus typischen Programmiersprachen übernommen und damit mit einer automatisiert ausführbaren Semantik unterlegt. Neben Kontrollstrukturen (wie Schleifen oder Fallunterscheidungen) kann ein Prozess analog zu Funktionen abgebrochen oder ein anderer aufgerufen werden. Im Detail gibt es leichte Unterschiede wie die Einschränkung, nur Subprozesse aufgerufen zu können. Andererseits gibt es auch neue Konstrukte wie dasjenige zum (Abbruch und) Neustart des aktuellen Prozesses. In typischen Programmiersprachen wäre dies nur durch Kombination mehrerer Konstruktionen nachbildbar.

Ablaufbedingung. Pro Aktivität ist die Abarbeitung der Anweisungen nur durch ihre Ordnung festgelegt. Auf Ebene der Aktivitäten können Vorbedingungen verwendet werden, um die Anweisungen einer Aktivität erst dann auszuführen, wenn die Vorbedingung erfüllt ist. Damit kann auf ein bestimmtes Ereignis reagiert werden. Auf Ebene der Prozesse werden die Ablaufbedingungen ausgewertet, die den Ablauf der Subprozesse eines Prozesses

bestimmen. Durch die entsprechenden Operatoren kann dabei eine beliebig sequentielle bzw. parallele Verschaltung definiert werden.

Ausnahmebehandlungen. Analog zu Programmiersprachen bietet JIL die Möglichkeit, Ausnahmen auszurufen (zu „werfen“) und diese durch Anweisungen zu behandeln.

Im Unterschied zu typischen Programmiersprachen werden die Behandlungen jedoch nicht innerhalb der Anweisungen eingewoben (vgl. *try-catch*-Blöcke in Java) – sondern neben den Aktivitäten. Um das Java-Beispiel fortzuführen, entspricht dies etwa der Situation, try-catch-Blöcke nur um den gesamten Funktionsrumpf ziehen zu können.

Reaktive Steuerung. Reaktionen sind Aktivitäten die auf verschiedene, vordefinierte Ereignisse reagieren, wie beispielsweise den erfolgreichen Abschluss eines (Sub-)Prozesses oder das Scheitern der Prüfung der Nachbedingung. Ein Interessantes Ereignis ist die Aktualisierung eines Parameters während der Ausführung eines Prozesses: Damit unterscheidet sich JIL zum Funktionskonzept typischer Programmiersprachen. Zwar kann bei einer Funktion durch Übergabe von Aktualparametern als Referenzen ebenfalls die Eingabe während der Funktionsausführung geändert werden. Doch es gibt keine expliziten Konstrukte, mit denen ein bestimmter Anweisungsblock ausgeführt wird, sobald dies eintritt. Ein solcher Effekt kann nur mit einer umfangreichen Kombination anderer Konstrukte nachgebaut werden. Zur Veranschaulichung zeigt Abbildung 3.5 zwei Beispiele des Einsatzes dieses Konstrukts in konkreter Syntax von JIL. Die Reaktion im linken Beispiel löst bei einer Änderung (Update) eines Anforderungsdokumentes (Requirements_File) aus. Die Behandlung besteht darin, dass der Subprozess Propagate_Requirements_Change aufgerufen wird. Jener Subprozess ist, an anderer Stelledefiniert (stellt also kein reserviertes Sprachkonstrukt von JIL dar). Im zweiten Beispiel (rechts im Bild) wird auf eine Änderung der Anforderungsspezifikation (Reqt_Spec) reagiert. Die Behandlung besteht hier darin, dass der Subprozess Identify_Classes_And_Objects abgebrochen wird.

```
REACT TO Requirements_File.Update BY
    INVOKE SUBPROCESS
        Propagate_Requirements_Change;
  END IF;
END REACT;
```

```
REACT TO UPDATE OF Reqts_Spec BY
    TERMINATE Identify_Classes_And_Objects;
END REACT;
```

Abbildung 3.5 Umgang mit Änderungen in einem Projekt in JIL. Quelle: [73,63]

Bewertung. Trotz der hilfreichen neuen Konstrukte wird die Definition eines Prozesses als aktualisierbare Transformation eines Projektstandes nicht vereinfacht: Die durch ein Projektteam vorgenommenen Änderungen treten zwar als Ereignisse auf (im Gegensatz zu Programmiersprachen muss deren Erkennung also nicht erst ausformuliert werden), die dazu passenden Reaktionen muss[23] ein Prozessingenieur jedoch selbst definieren.

3.2.2 XML Process Definition Language (XPDL 2.1)

Die Prozessbeschreibungssprache XPDL (XML Process Definition Language) [74] wurde von der WfMC (Workflow Management Coalition) [75] als Standard für die formale Definition von Geschäftsprozessen eingeführt, kann aber auch für Vorgehensmodelle

[23] Zum Vergleich: In der Programmiersprache Java wird einem Entwickler die Arbeit abgenommen, sich um die Freigabe reservierten Speichers zu kümmern.

eingesetzt werden (vgl. [76]). Sie wird u.a. als abstrakte Syntax[24] der BPMN (Business Process Modelling Notation) [77] verwendet, die wiederum von der OMG definiert ist. Abbildung 3.6 zeigt die sinngemäße, abstrakte Syntax von XPDL.

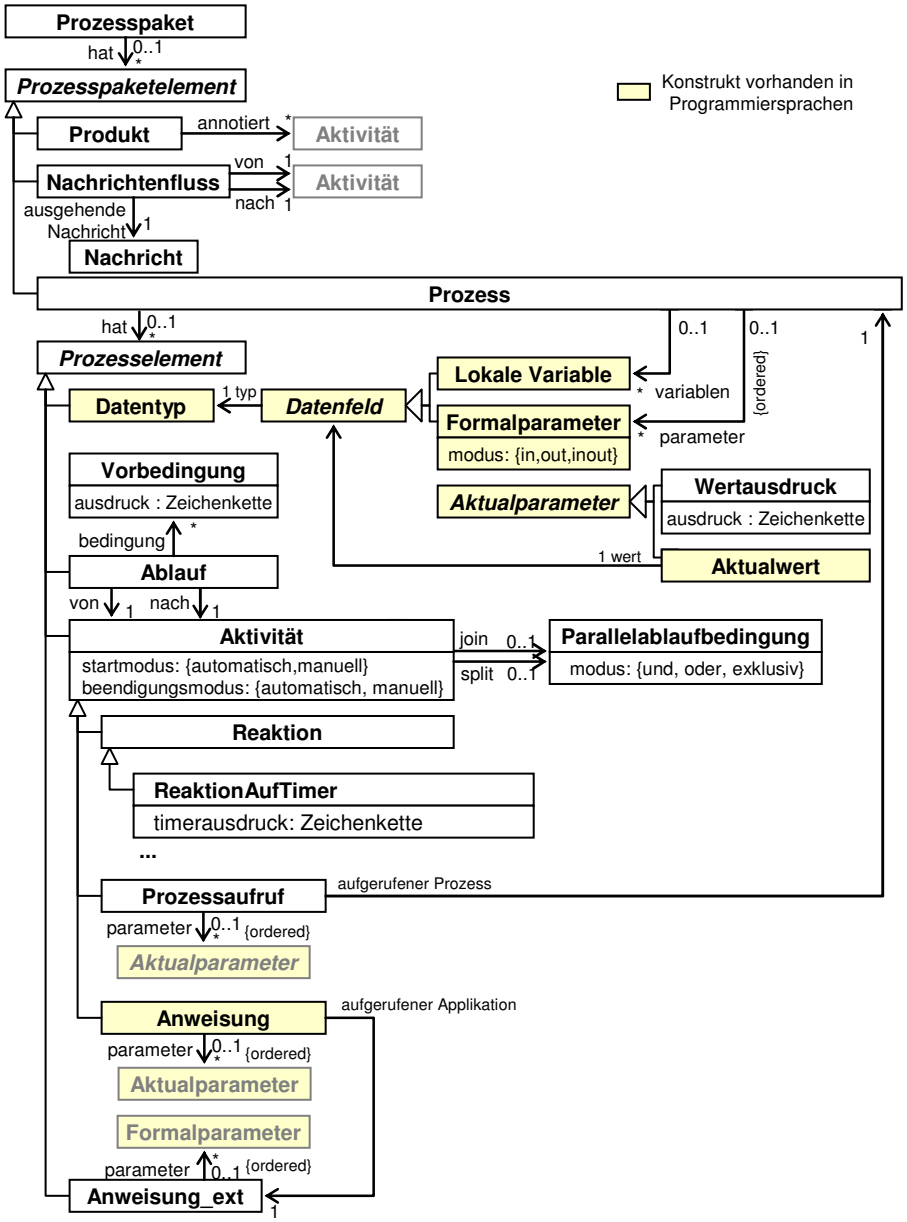


Abbildung 3.6 XPDL, abstrakte Syntax, sinngemäß

Prozess und Aktivität. Ein Prozess in XPDL wird auch als Workflow(-Prozess) bezeichnet und entspricht vom Prinzip her einem Prozess aus JIL. Während in JIL ein Prozess hierarchisch in Subprozesse gegliedert werden konnte, liegen bei XPDL alle Prozesse auf gleicher Ebene nebeneinander. Eine hierarchische Gliederung ist über Prozesspakete nur bis zu einer Ebene möglich. Ansonsten verfügt ein Prozess in XPDL ebenfalls über Formalparameter und kann aus anderen Prozessen heraus aufgerufen werden. Die Aufrufe erfolgen dabei allerdings nicht mehr durch Anweisungen, sondern durch (spezielle) Aktivitäten. Allgemein sind Anweisungen und Aktivitäten gegenüber JIL verschmolzen. So

[24] Die in Abbildung 3.6 gezeigte Syntax – in Form eines Modells – ist damit die abstrakte Syntax der abstrakten Syntax der BPMN. Die konkrete Syntax von XPDL, und gleichzeitig die abstrakte Syntax von BPMN, ist ein XML-Derivat.

verschwinden Schleifen und Fallunterscheidungen – die in XPDL mit Ablaufbedingungen zwischen Aktivitäten ausgedrückt werden.

Ablaufbedingung. Sequenzen werden über einen einzelnen Ablauf zwischen zwei Aktivitäten beschrieben. Parallelabläufe werden durch mehrere von einer Aktivität ausgehender Abläufe gestartet, und durch mehrere zu einer Aktivität eingehender Abläufe wieder beendet. Vorbedingungen sind hier direkt an Abläufe gehängt – im Unterschied zu JIL, wo diese direkt an den Aktivitäten hängen.

Reaktive Steuerung. Reaktionen werden in XPDL als Spezialfall von Aktivitäten betrachtet. Die konkreten Reaktionsarten sind von der Art her mit denen aus JIL vergleichbar. Eine Besonderheit besteht allerdings in der Erzeugung und Zustellung von Ereignissen: Ein Nachrichtenfluss beschreibt welche Aktivität (Assoziation von) welche Nachricht (Assoziation ausgehende Nachricht) erzeugt, und welche Aktivität (Assoziation nach) auf deren Eingang wartet.

Extern definierte Berechnung. Vorbedingungen und Ausdrücke werden durch Zeichenketten abgebildet, wodurch deren Syntax nicht mehr im Rahmen von XPDL liegt. Allgemein können beliebige Berechnungen durch eine externe Anweisung (Konstrukt Anweisung_ext[25]) ausgelagert werden. Für XPDL sind nur deren Formalparameter bekannt, die bei deren Anwendung (Konstrukt Anweisung) mit Aktualparametern aus dem Prozess belegt werden können.

Produkt und Prozesspaket. Ein Produkt beschreibt die Art eines im Projekt ggf. zu erstellenden Ergebnisses. Zwei Beispiele sind „Vertrag“ und „SW-Modul“. Produkte werden nicht als Teil von Prozessen, sondern außerhalb als Teil von Prozesspaketen definiert.

In XPDL sind Produkte mit keiner formalen Semantik unterlegt, auch wenn diese einzelnen Aktivitäten zugeordnet werden können. Eine solche Zuordnung beschreibt nur informal, dass bei der entsprechenden Aktivität das zugeordnete Produkt entweder als Eingabe verwendet wird, oder als Ausgabe entsteht. Produkte könnten zwar durch Datenfelder formal fassbar gemacht werden. Doch Datenfelder sind in erster Linie nicht zum Vorhalten und Transportieren von Projektergebnissen gedacht, sondern ganz allgemein für beliebige Berechnungen und damit verbundene Zwischenergebnisse (vgl. typische Programmiersprache).

Bewertung. Im Bezug auf die Möglichkeit zur inkrementellen Transformation bietet XPDL gegenüber JIL keine Vorteile. Durch den größeren Anteil der für Prozessbeschreibungssprachen dedizierter Konstrukte ist zumindest ein höheres Abstraktionsniveau zu konstatieren: Insbesondere für die Definition der Ablaufbedingungen ähnelt XPDL den UML-Aktivitätsdiagrammen, welche in der Praxis inzwischen eine breite Akzeptanz gefunden haben.

3.2.3 Software & Systems Process Engineering Meta-Model (SPEM 2.0)

SPEM [3] ist von der OMG als Prozessbeschreibungssprache für die Definition von Vorgehensmodellen eingeführt. Sie wird u.a. für die Definition des RUP (Rational Unified Process) [78] und für HERMES [79] – des schweizerischen Vorgehensmodells der Bundesverwaltung – verwendet. Abbildung 3.7 zeigt die sinngemäße, abstrakte Syntax von SPEM.

[25] Im Original als Application bezeichnet.

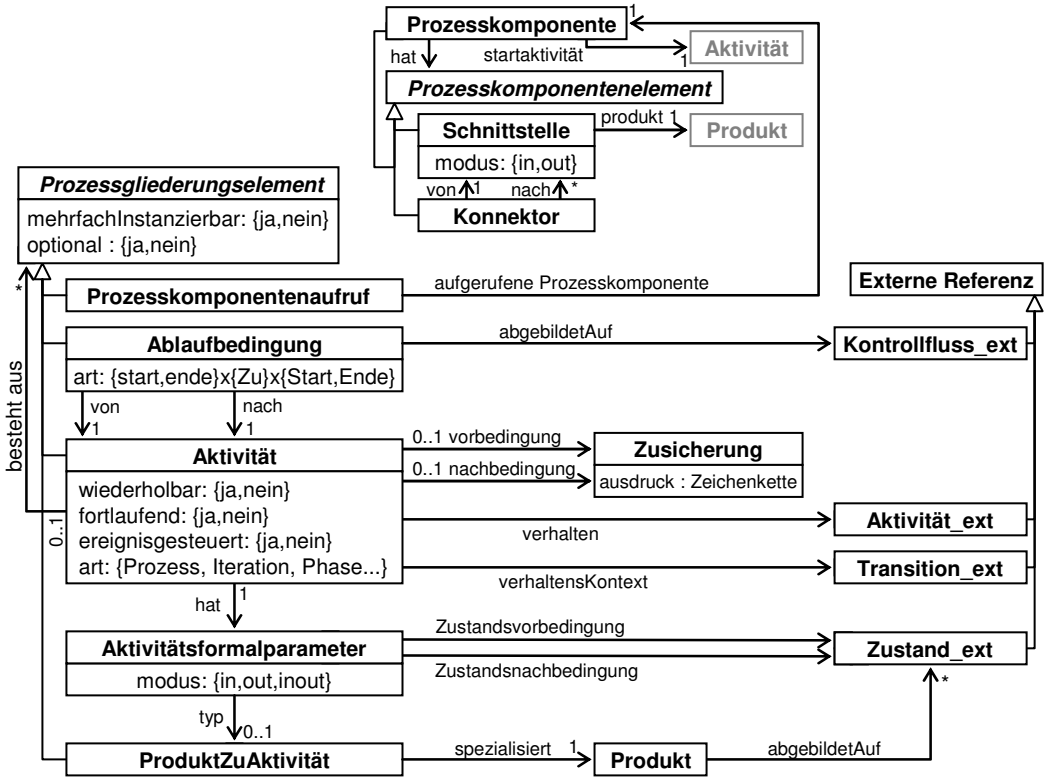


Abbildung 3.7 SPEM, abstrakte Syntax, sinngemäß

Prozess, Aktivität und Produkt. In SPEM wird syntaktisch nicht zwischen einem Prozess und einer Aktivität unterschieden. Stattdessen werden beide als Aktivität gehandhabt. Sofern für einen Prozessingenieur diese Unterscheidung wichtig ist, kann diese durch Belegung des Attributs art vorgenommen werden. Neben dem Begriff des Prozesses stehen noch weitere wie Iteration und Phase zur Verfügung. Entsprechend der Zusammenfassung des Prozess- und Aktivitätsbegriffs sind Aktivitäten hierarchisch und besitzen Formalparameter. Im Unterschied zu XPDL handelt es sich bei den Formalparametern jedoch nicht um Steuerungsinformationen, sondern direkt um Projektergebnisse: die Produkte.

Bei Produkten wird zwischen zwei Definitionsstufen unterschieden: Eine Instanz des Konstrukts Produkt enthält eine universell geltende Beschreibung. Eine Instanz des Konstrukts ProduktZuAktivität ergänzt eine solche Beschreibung um aktivitätsspezifische Details. So kann beispielsweise die Optionalität eines Produktes, oder die Frage, wie viele Instanzen es zu einem Produkt geben kann, pro Aktivität (genauer: pro eines ihrer Formalparameter) festgelegt werden.

Der Ablauf wird durch Ablaufbedingungen über Aktivitäten definiert. Diese gleichen denen aus XPDL – mit dem Unterschied, dass Sequenzen und Parallelabläufe durch die vier Beziehungsarten Start-zu-Ende, Ende-zu-Start, Start-zu-Start und Ende-zu-Ende ausgedrückt werden.

Prozesskomponente. In SPEM ermöglichen Prozesskomponenten die einfache Komposition und – durch Verwendung von Schnittstellen – auch eine einfache Austauschbarkeit von Prozessteilen. Eine Schnittstelle definiert jeweils ein Produkt welches entweder benötigt oder angeboten wird. Die Verschaltung von Komponenten erfolgt mittels Konnektoren. Sofern die Verschaltung innerhalb einer hierarchischen Komponente erfolgt, können – wie von Komponentensystemen durch die Delegation (vgl. [80]) gewohnt – die Schnittstellen der äußeren Komponente mit denen der inneren Komponenten verschaltet werden. Ein Vorgehensmodell muss dabei nicht durchgängig aus Prozesskomponenten zusammengesetzt

werden. So kann eine Komponente auch aus einer beliebigen Stelle einer hierarchischen Aktivität heraus aufgerufen werden (Konstrukt Prozesskomponentenaufruf).

Extern definierte Berechnungen und Verhalten. Während in XPDL nur einzelne Berechnungen ausgelagert wurden, wird in SPEM auch der Aufruf derselben ausgelagert. Allgemein kann in SPEM sogar ein beliebiges Verhalten ausgelagert – bzw. dem beschriebenen Prozess (bzw. Aktivität) unterlegt werden. Möglich wird dies dadurch, dass die Elemente eines Prozesses (bzw. einer Aktivität) auf Eckpunkte eines z.B. mit UML-Aktivitätsdiagrammen extern definierten Verhaltens abgebildet werden. Als Eckpunkte dienen dabei UML-Aktivitäten (Konstrukt Aktivität_ext), Transitionen (Konstrukt Aktivität_ext) und Kontrollflüsse (Konstrukt Kontrollfluss_ext). Dabei wird nicht gefordert, dass alle Elemente des extern definierten Verhaltens abgedeckt sind. Somit kann zwischen den gewählten Eckpunkten ein beliebiges Verhalten liegen.

Bewertung. Obwohl SPEM eine Reihe von Konstrukten für eine flexible Definition eines Vorgehensmodells bietet, können damit, wie schon mit JIL und XPDL, keine Vorgaben definiert werden, die in Form einer inkrementellen Transformation angewendet werden können. Ein Prozessingenieur muss also auch hier den Umgang mit Änderungen ausformulieren.

3.2.4 V-Modell XT 1.3 Metamodell (VM³, PBS_0)

Das V-Modell XT 1.3 Metamodell ist eine Prozessbeschreibungssprache die im Rahmen der Entwicklung des Vorgehensmodells V-Modell XT entstanden ist, und selbigem eine formale Beschreibungsgrundlage gibt. Gleichwohl ist sie auch für Varianten (z.B. V-Modell XT Bayern [81], V-Modell XT Bund [82], flyXT [83] oder das V-Modell von Witt-Weiden [84], um einige zu nennen), bzw. im Allgemeinen für beliebige Vorgehensmodelle nutzbar. Abbildung 3.8 zeigt die sinngemäße, abstrakte Syntax. Für eine detaillierte Abhandlung sei auf [85] verwiesen.

Diese Prozessbeschreibungssprache wird im Weiteren der Arbeit für ein durchgängiges Beispiel herangezogen. Um die tatsächliche abstrakte Syntax des V-Modell XT 1.3 Metamodells von der sinngemäßen in Abbildung 3.8 zu unterscheiden, wird erstere als VM³ und letztere als PBS_0 bezeichnet[26].

[26] PBS steht für *P*rozess*b*eschreibungs*s*prache, 0 für die Ausgangssituation.

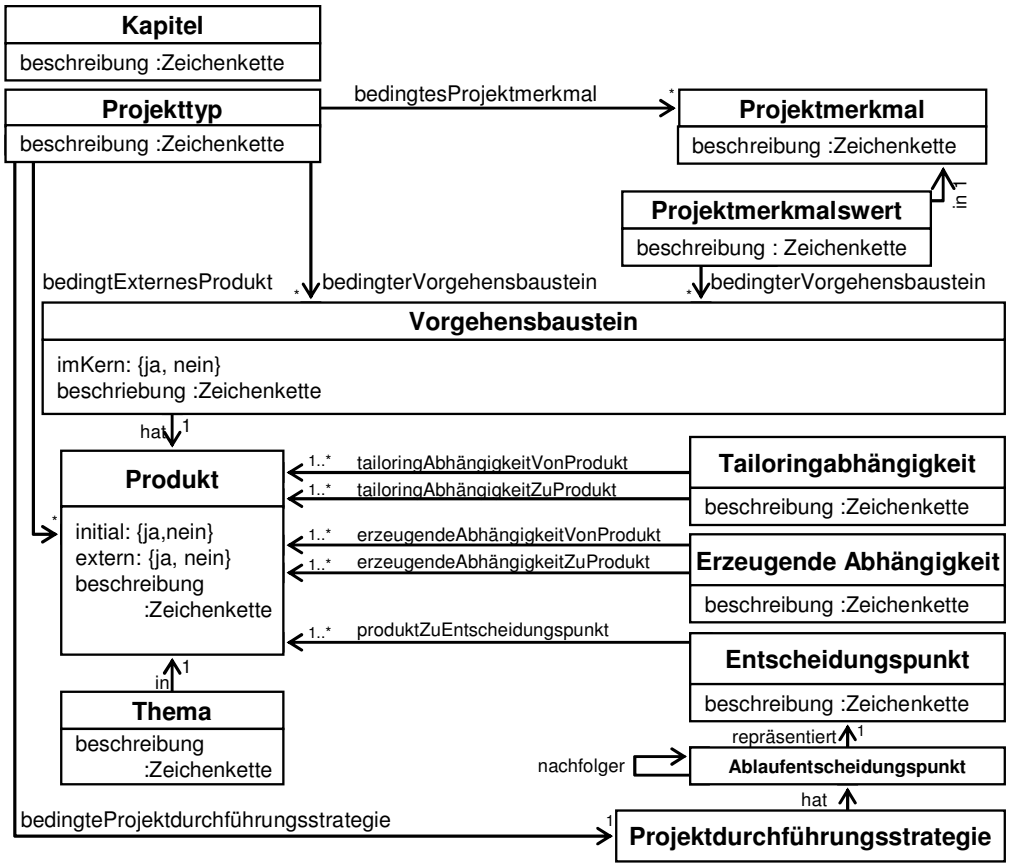


Abbildung 3.8 V-Modell Metamodell 1.3 (PBS_0), abstrakte Syntax, sinngemäß

Produkt und Inhalt. Die in einem Projekt zu erarbeitenden Ergebnisse werden in PBS_0 als Produkte bezeichnet und können in Themen gegliedert werden. Konkrete Beschreibungen der jeweils zu erstellenden Inhalte werden jedoch nicht weiter strukturiert, sondern nur als Zeichenketten behandelt (Attribut beschreibung). Ein Produkt kann initial sein, was bedeutet, dass es im Projekt genau ein Dokument mit diesem Typ geben muss. Ein Produkt kann extern sein, was bedeutet, dass es nicht innerhalb des Projektes erstellt, sondern von Außen an das Projekt herangetragen wird. Dennoch wird die Struktur solcher Produkte vom Vorgehensmodell definiert, um die Inhalte korrekt interpretieren zu können.

Tailoring. Das Tailoring ist ein Mechanismus zur Anpassung des Vorgehensmodells und beruht auf der Auswahl von für das Projekt relevanten Teilen aus der vordefinierten Menge. Dazu werden Produkte durch die beiden Ebenen Projekttyp und Vorgehensbausteine modularisiert. Auf der ersten Ebene ist genau einer der möglichen Projekttypen zu wählen. Dadurch werden relevante Vorgehensbausteine bestimmt (Assoziation bedingterVorgehensbaustein des Konstrukts Projekttyp), die wiederum fachlich zusammengehörende Produkte umfassen. Aus der Wahl eines Projekttyps ergeben sich zusätzlich die mit einem Projektmerkmalswert zu belegende Projektmerkmale (Assoziation bedingtesProjektmerkmal). Je nach gewähltem Projektmerkmalswert werden weitere Vorgehensbausteine relevant (Assoziation bedingterVorgehensbaustein des Konstrukts Projektmerkmalswert). Vorgehensbausteine die im sogenannten Vorgehensmodell kern liegen (Attribut imKern) sind dagegen immer relevant. Dagegen kann die Externheit eines Produktes auch in Abhängigkeit des Projekttyps definiert sein. Durch die Assoziation bedingtExternesProdukt kann ein im Allgemeinen nicht-externes Produkt bei der Wahl eines Projekttyps als für diesen extern definiert werden[27].

[27] Dies ist eine gegenüber VM³ kompaktere Darstellung. In VM³ gibt es eine solche Assoziation nicht. Stattdessen müssen und werden bei der Anwendung jeweils zwei Produkte angelegt – eines als externes, und

Dynamisches Tailoring. Das Dynamische Tailoring unterscheidet sich vom Tailoring dahingehend, dass zur Auswahl der relevanten Teile die erarbeiteten Ergebnisse ausgewertet werden. Entsprechend der über den Projektzeitraum sich stetig verändernden Menge der Ergebnisse ist das Dynamische Tailoring kontinuierlich durchzuführen – und nicht nur zu Projektbeginn. Die dabei anzuwendenden Regeln werden durch das Konstrukt Tailoringabhängigkeit abgelegt. Formal greifbar sind dabei nur die betroffenen Produkte – nicht jedoch die Bedingungen oder die konkret vorzunehmenden Änderungen des Vorgehensmodells. Diese sind nur informal über das Attribut beschreibung abgelegt.

Induktiver Ergebnisumfang. Die konkrete Menge der Ergebnisse für ein Projekt ist induktiv bestimmt. Den Induktionsanfang bilden die initialen Produkte, zu denen jeweils genau ein Ergebnis zu erstellen ist. Alle weiteren im Projekt zu erstellenden Ergebnisse ergeben sich durch das Befolgen der erzeugenden Produktabhängigkeiten, die Induktionsschritten entsprechen. Formal greifbar sind mit PBS_0 nur die betroffenen Produkte – nicht jedoch die Bedingungen oder die diesbezüglich vorzunehmenden Einplanungen (vgl. Attribut beschreibung). Zusätzliche Induktionsanfänge ergeben sich durch die externen Produkte, die die Erzeugung nicht-externer Produkte verursachen können (z.B. Fehlermeldung).

Projektablauf. Ein Projektablauf wird von PBS_0 als eine Folge von Meilensteinen angesehen, wobei jeder Meilenstein durch einen Entscheidungspunkt getypt sein kann. Ein Entscheidungspunkt legt über die Assoziation produktZuEntscheidungspunkt fest, welche Ergebnisse aus dem Ergebnisumfang zum zugehörigen Meilenstein fertig gestellt sein müssen. In welcher Reihenfolge die Meilensteine eines Projektes mit Entscheidungspunkten getypt sein dürfen bzw. müssen, ergibt sich aus der Projektdurchführungsstrategie, die wiederum eindeutig durch die Wahl des Projekttyps bestimmt ist. Um Entscheidungspunkte wiederzuverwenden (sowohl für verschiedene Projektdurchführungsstrategien als auch innerhalb einer Projektdurchführungsstrategie mit verschiedenen Nachfolgern) ist die Nachfolgerbeziehung nicht direkt auf Entscheidungspunkten definiert, sondern auf Ablaufentscheidungspunkten[28]. Insgesamt ergibt sich durch die Projektdurchführungsstrategien eine Fertigstellungsreihenfolge für den Ergebnisumfang. D.h. Arbeiten an Ergebnisse für spätere Meilensteine können bereits in früheren Meilensteinen begonnen werden – es wird nur festgelegt, in welcher Reihenfolge die Ergebnisse zu fixieren sind.

Kapitel. Das Konstrukt Kapitel kann genutzt werden, um querschnittlich geltende informale Vorgaben abzulegen. Dies sind also solche, die einerseits nicht zur Semantik von PBS_0 gehören, andererseits aber auch nicht genau einer Instanz eines anderen Konstrukts zugeordnet werden können.

Bewertung. Das PBS_0 ist vollständig deklarativ, d.h. sämtliche formalen Berechnungsvorschriften sind bereits Teil der fixierten Semantik, und können von einem Prozessingenieur nicht erweitert werden. Diese Semantik basiert jedoch nicht auf einer inkrementellen Transformation. Außerdem können einige Vorgaben zur Planung nur informal abgelegt werden.

---

eines als internes. Beispielsweise gibt es die „Lieferung“ als nicht-externes sowie dessen Gegenstück „Lieferung (von AN)“ als externes Produkt. Da beide Produkte folglich nicht in einem Vorgehensbaustein vorkommen dürfen, sind entsprechende Schnitte der Vorgehensbausteine notwendig.

[28] Dies ist eine gegenüber VM³ vereinfachte Darstellung. VM³ erlaubt Parallelitäten und besitzt eine Modulare Struktur der Abläufe (vgl. [85]), die u.a. in Abhängigkeit der gewählten Projektmerkmale steht.

3.3 VM_0 – Ein einfaches Vorgehensmodellbeispiel

In diesem Kapitel wird das Vorgehensmodell VM_0 definiert, welches ein Ausschnitt des Vorgehensmodells V-Modell XT in der Version 1.3 [86] ist. Der Ausschnitt ist so gewählt, dass einerseits ein Projekt durchgängig damit beschrieben werden kann. So wird der gesamte Verlauf – von der Genehmigung bis zum Projektabschluss – abgedeckt. Andererseits wird der Ausschnitt auch so gewählt, dass darin interessante Vorgehensmodellkonzepte wie Tailoring, dynamisches Tailoring und induktiver Ergebnisumfang durch mindestens ein Beispiel abgedeckt sind. Die für den Ausschnitt gewählten Elemente werden in ihren Beschreibungstexten auf das Minimale gekürzt. Die Abbildung 3.9 und Abbildung 3.10 zeigen den resultierenden Ausschnitt im Überblick. Die erste Abbildung zeigt die Entscheidungspunkte und die zugehörigen Produkte, die zweite die beiden Projekttypen. In den Unterkapiteln 3.3.1 – 3.3.5 werden die Bestandteile von VM_0 als Instanz der Prozessbeschreibungssprache PBS_0 und im Detail dargestellt.

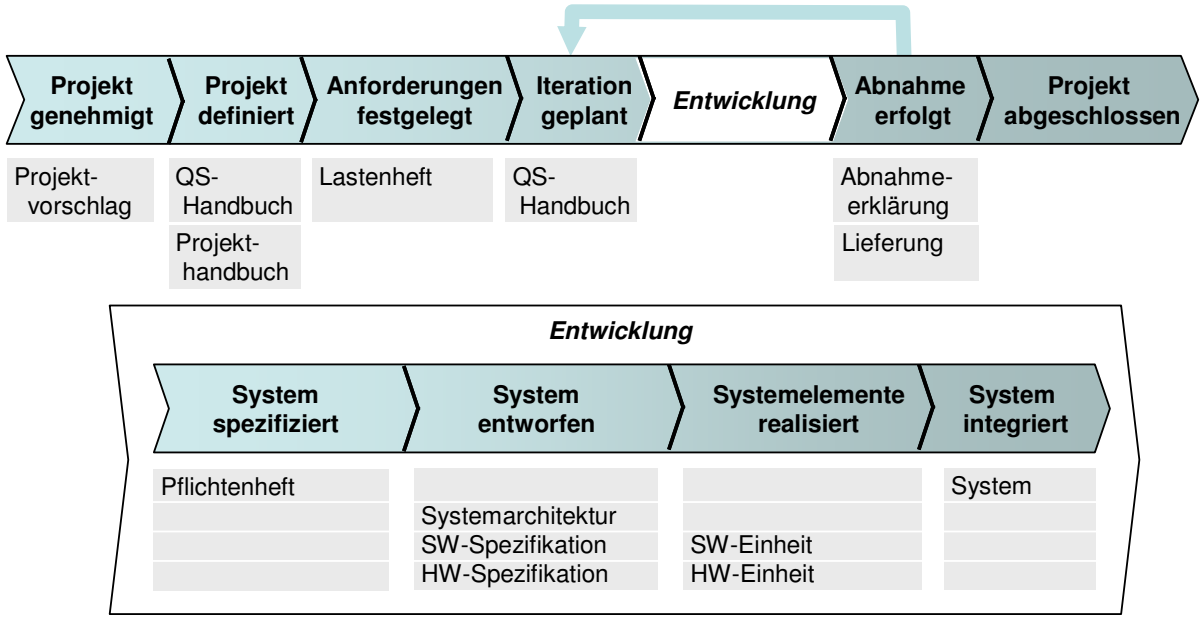


Abbildung 3.9 Überblick über Entscheidungspunkte und Produkte in VM_0

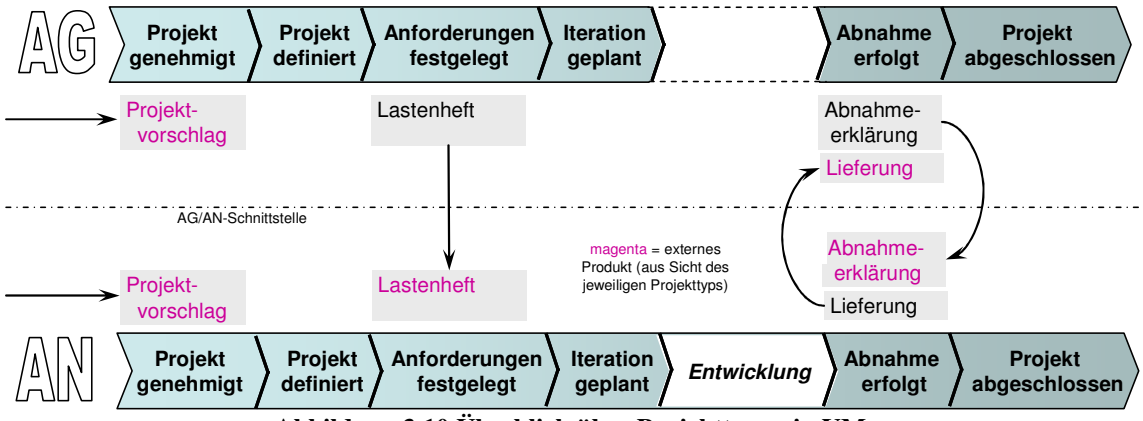


Abbildung 3.10 Überblick über Projekttypen in VM_0

3.3.1 Produkte und Inhalte

Abbildung 3.11 und Abbildung 3.12 zeigen alle Produkte und Themen von VM_0. Themen sind dabei in Verbindung mit den Produkten dargestellt, zu denen sie gehören. Die Aufteilung der Produkte auf die beiden Abbildungen ist an den beiden „Ebenen“ in Abbildung 3.9 ausgerichtet (mit den Produkten für die Entwicklung in Abbildung 3.12).

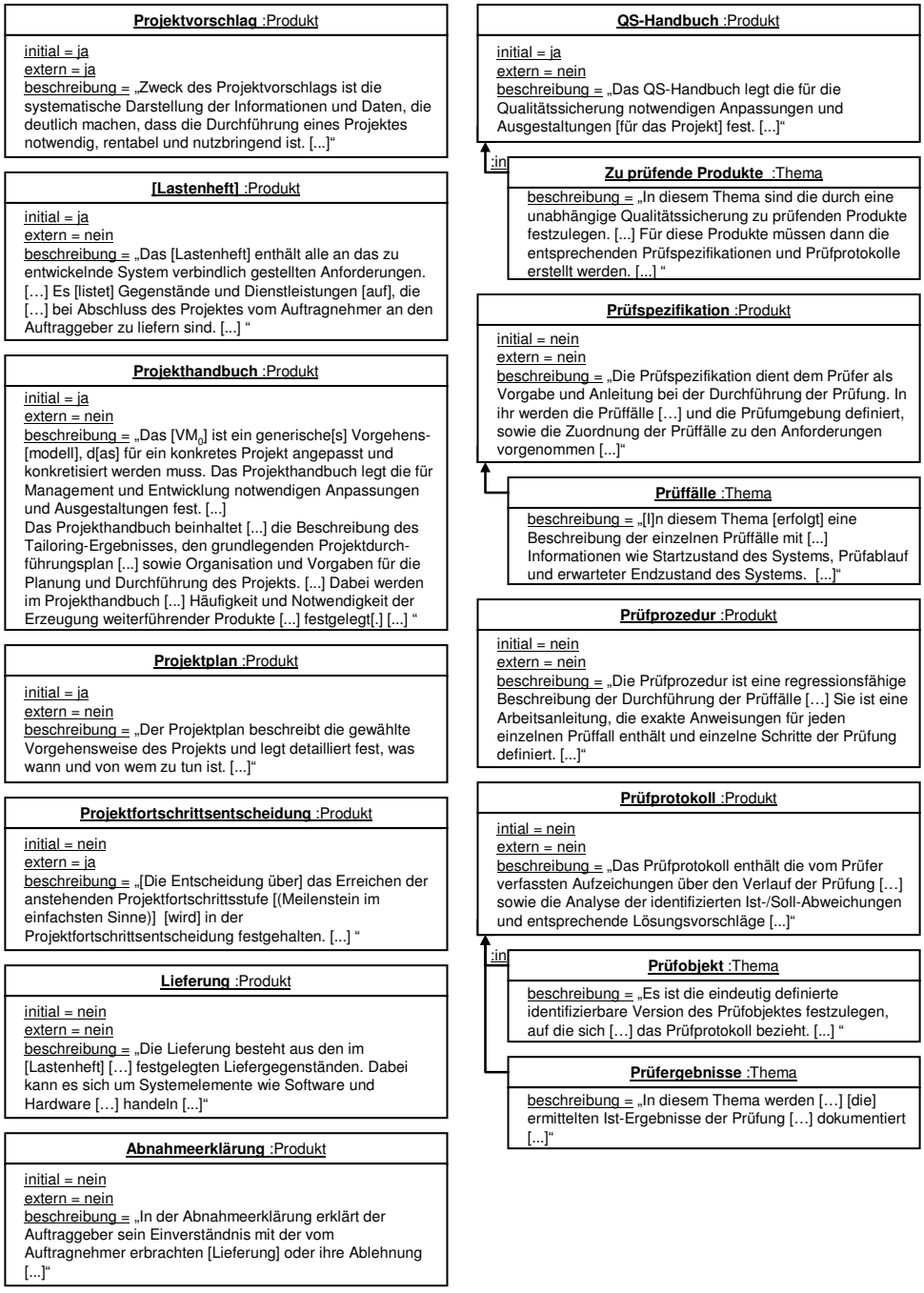


Abbildung 3.11 Produkte und Themen im VM_0 als Instanz von PBS_0 (Teil 1/2)

Den Produkten QS-Handbuch und Projekthandbuch (Abbildung 3.11) kommt eine besondere Rolle zu: sie gestalten das in einem Projekt verwendete Vorgehensmodell aus. Dies kann dahingehend verallgemeinert werden, dass Produkte (bzw. Ergebnisse) auch ein vollständiges Vorgehensmodell enthalten können. Auch dem Projektplan (Abbildung 3.11) kommt eine besondere Rolle zu: Durch das Definieren des Solls kann ein entsprechendes Ergebnis reflektiv über das Projekt (in dem es sich befindet) sprechen, und damit nicht nur die Erzeugung weiterer Ergebnisse, sondern auch seine eigene Überarbeitung einplanen. In jedem Falle muss ein Projektplan die Produkte des Vorgehensmodells kennen um diese den eingeplanten Ergebnissen zuzuordnen.

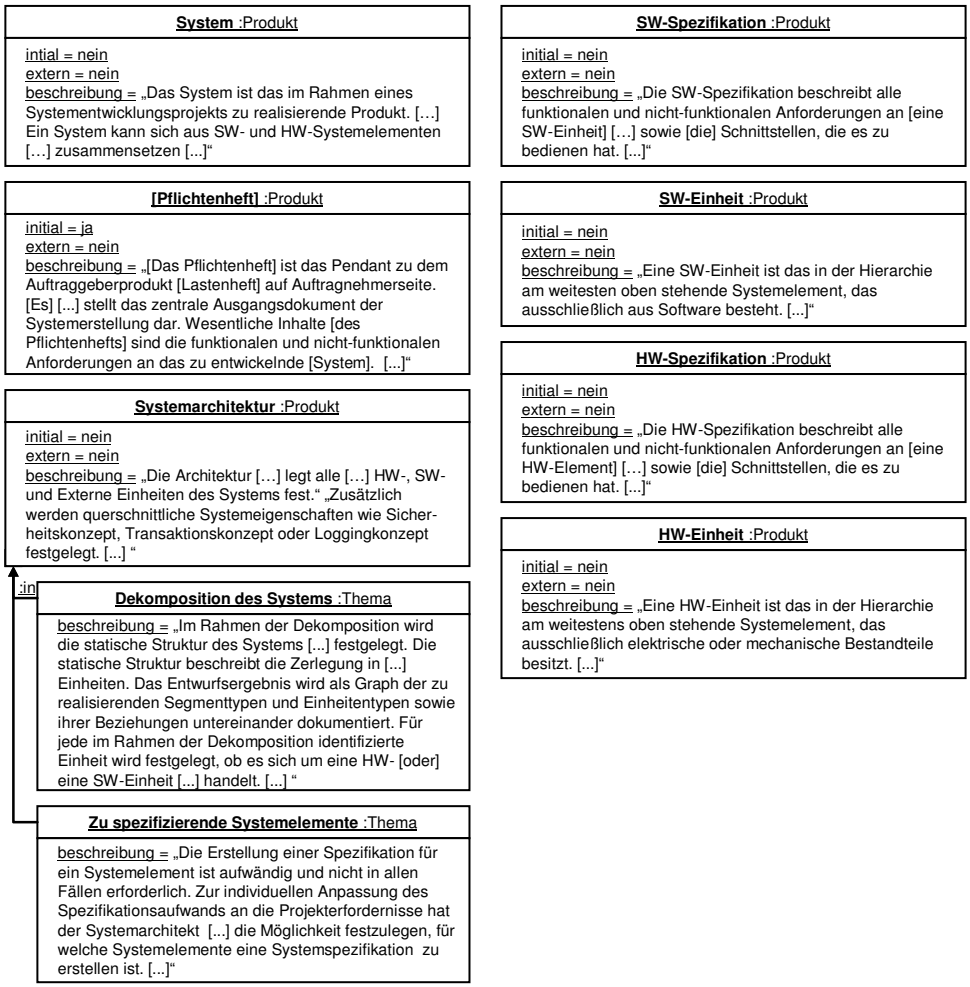


Abbildung 3.12 Produkte und Themen im VM_0 als Instanz von PBS_0 (Teil 2/2)

3.3.2 Tailoring

Abbildung 3.13 definiert die sieben Vorgehensbausteine von VM_0, die die Produkte aus Abbildung 3.11 und Abbildung 3.12 disjunkt aufteilen. Im Kern befinden sich nur die beiden Vorgehensbausteine Qualitätssicherung und Projektmanagement.

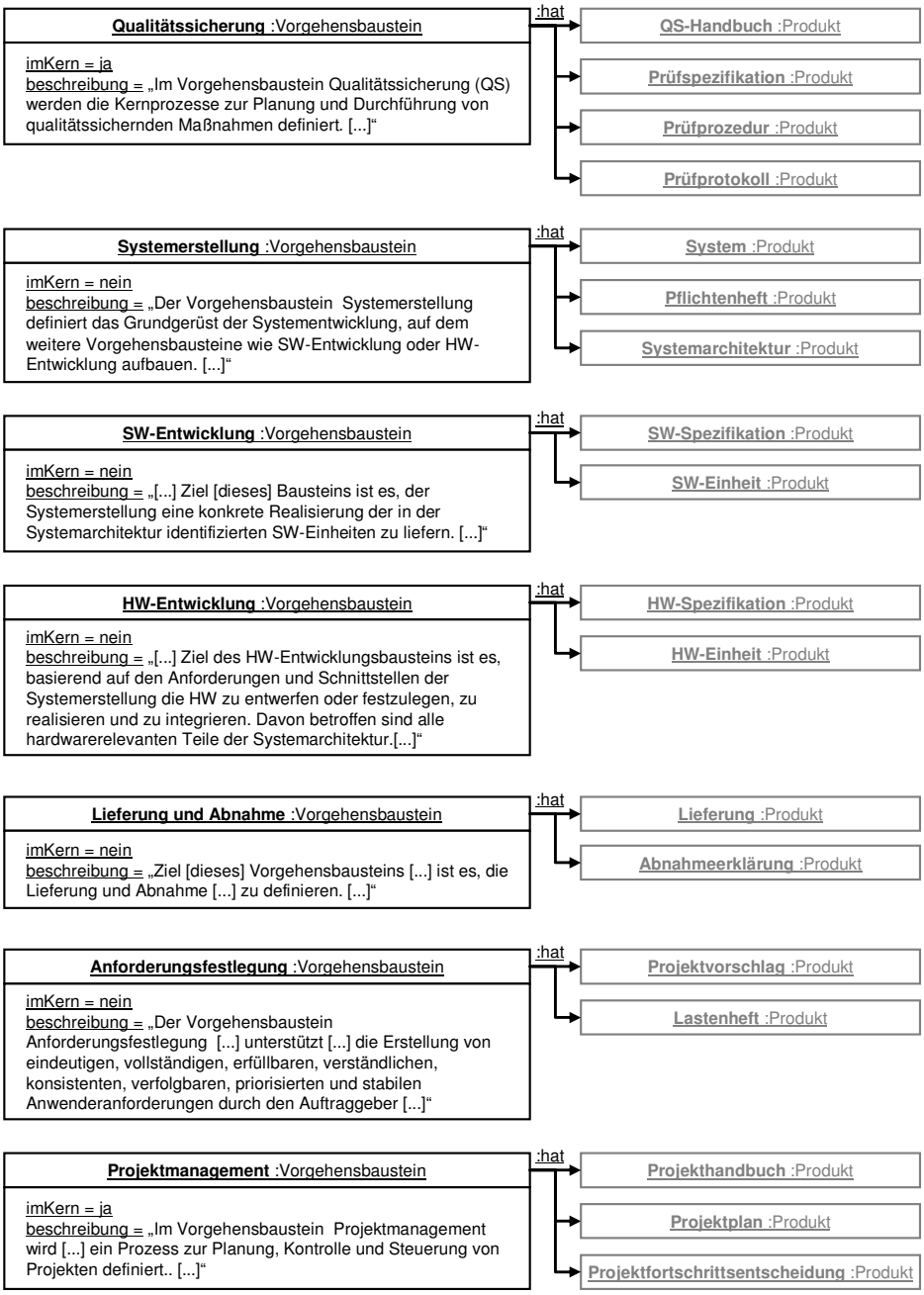


Abbildung 3.13 Vorgehensbausteine in VM_0 als Instanz von PBS_0

Abbildung 3.14 definiert die beiden Projekttypen von VM_0: Das Systemerstellungsprojekt eines Auftraggebers, und das Systementwicklungsprojekt eines Auftragnehmers. Für den ersten sind (neben denen des Kerns) die Vorgehensbausteine Anforderungsfestlegung und Lieferung und Abnahme relevant. Für den zweiten kommt mindestens der Vorgehensbaustein Systemerstellung hinzu. Weitere relevante Vorgehensbausteine ergeben sich je nach Wahl eines Projektmerkmalswerts für das Projektmerkmal Projektgegenstand (siehe Abbildung 3.15). Die Lieferung ist nur für das Systemerstellungsprojekt, das Lastenheft und die

Abnahmeerklärung nur für ein Systementwicklungsprojekt extern. Schließlich ist jedem Projekttyp eine eigene Projektdurchführungsstrategie zugeordnet.

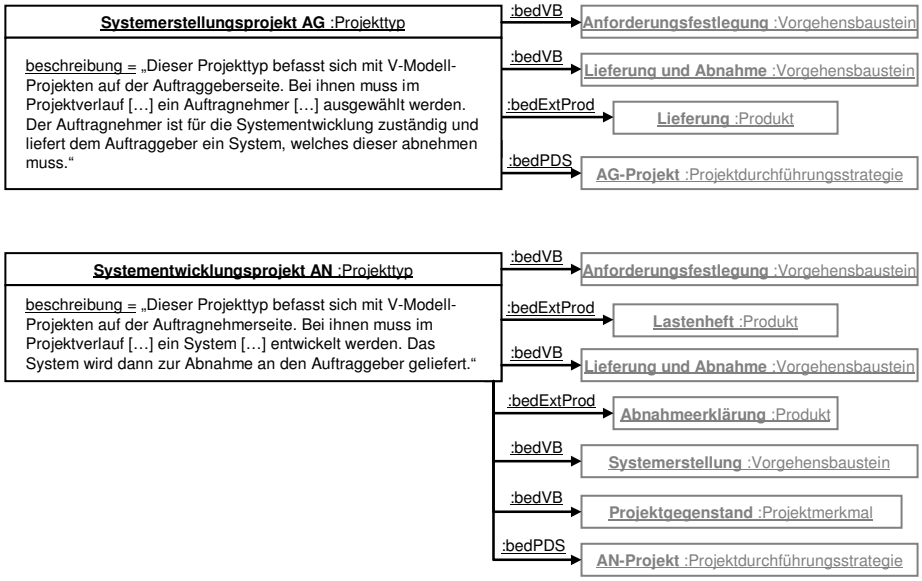


Abbildung 3.14 Projekttypen im VM_0 als Instanz von PBS_0

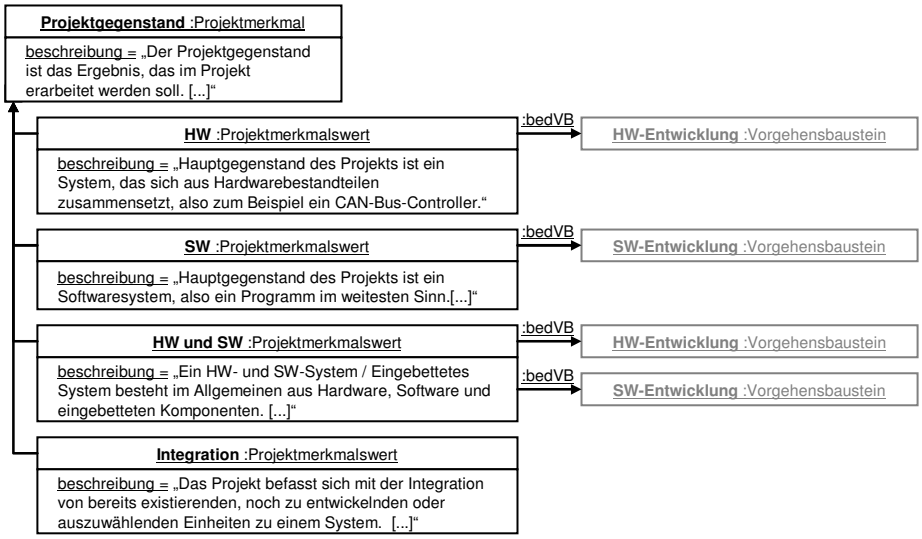


Abbildung 3.15 Projektmerkmale im VM_0 als Instanz von PBS_0

3.3.3 Dynamisches Tailoring

Abbildung 3.16 zeigt die einzige Tailoringabhängigkeit von VM_0 (die allerdings auch auf mehrere einzelne aufgeteilt werden könnte).

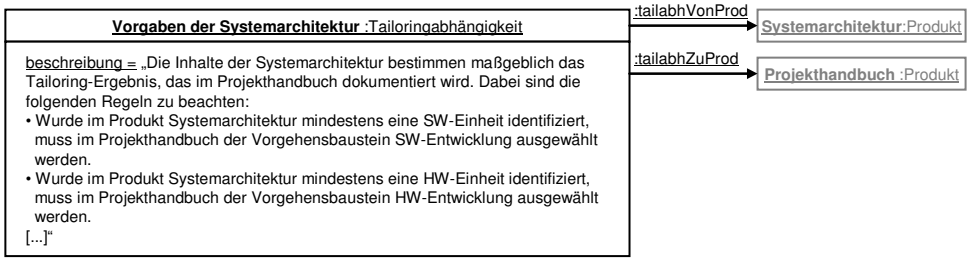


Abbildung 3.16 Tailoringabhängigkeit(en) im VM_0 als Instanz von PBS_0

3.3.4 Induktiver Ergebnisumfang

Abbildung 3.17 zeigt die erzeugenden Abhängigkeiten von VM_0. Der Kürze wegen wird auf eine zum Produktumfang einer SW-Einheit im System analoge Abhängigkeit für HW verzichtet.

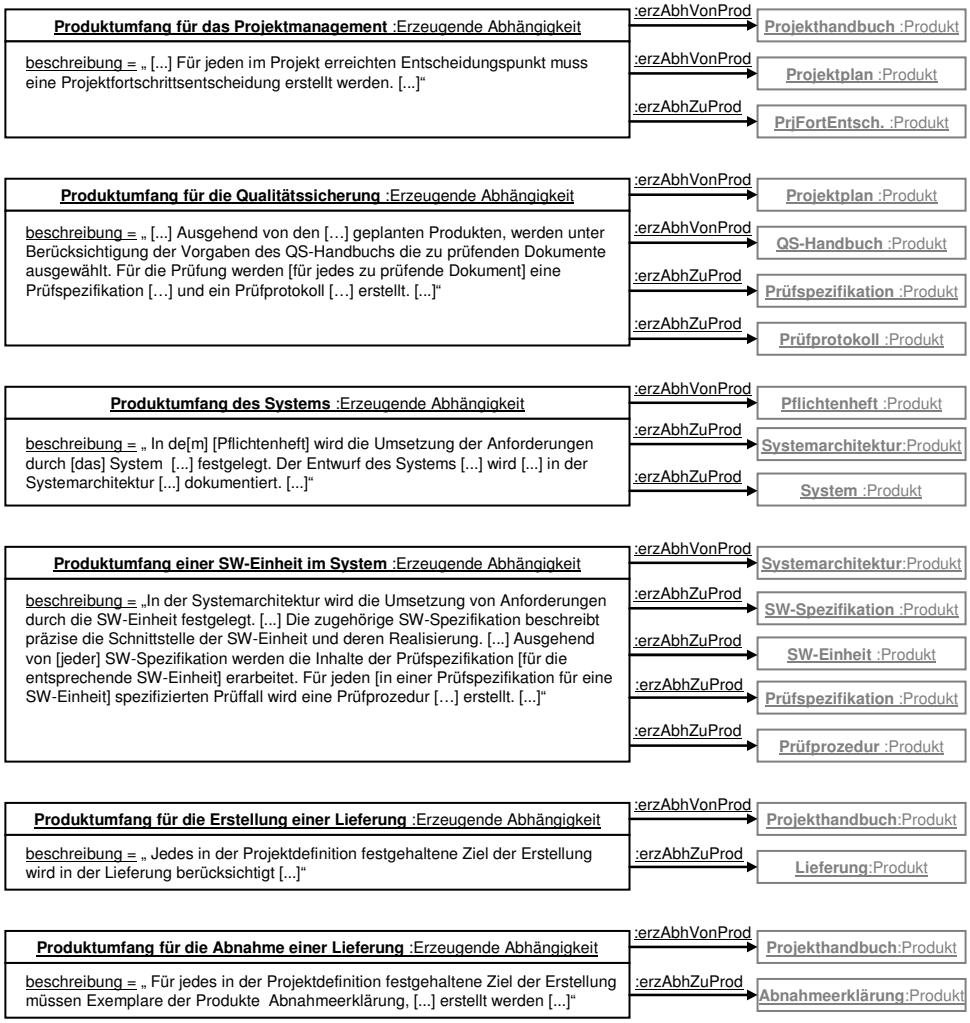


Abbildung 3.17 Erzeugende Produktabhängigkeiten im VM_0 als Instanz von PBS_0

Abbildung 3.18 zeigt eine querschnittliche Vorgabe die sich sowohl auf den Ergebnisumfang als auch auf den Umfang der Ergebnisversionen bezieht. So ist konkret bei jeder Vorlage zur eigenständigen Qualitätssicherung ein Prüfprotokoll zu erstellen.

1.4.5 Qualitätssicherung und Produktzustandsmodell :Kapitel

beschreibung = „ [...] Jedes [Ergebnis] besitzt einen Produktzustand. Mögliche Produktzustände sind in Bearbeitung, vorgelegt und fertig gestellt […]. Der Zustand eines [Ergebnisses] wird spätestens mit der erfolgreichen Beendigung der bearbeitenden Aktivität neu ermittelt. Um eine Aktivität erfolgreich zu beenden, muss das erzeugte [Ergebnis] entsprechend geprüft werden. […] Ist [dazu] eine eigenständige Qualitätssicherung notwendig, so wechselt das [Ergebnis] zuerst in den Zustand vorgelegt und nach der erfolgreichen Prüfung in den Zustand fertig gestellt. Andernfalls geht das [Ergebnis] sofort nach der erfolgreichen Eigenprüfung in den Zustand fertig gestellt über. Ist eine Prüfung nicht erfolgreich, so muss das [Ergebnis] entsprechend überarbeitet und erneut qualitätsgesichert werden. Sind dabei relevante Produktabhängigkeiten verletzt, so sind die Produktverantwortlichen der beteiligten [Ergebnisse] für die Beseitigung der Inkonsistenz verantwortlich. Dies kann auch dazu führen, dass die [V]erantwortlichen [...] entscheiden, dass ein bereits fertig gestelltes [Ergebnis] wieder in den Zustand in Bearbeitung gesetzt wird, um die notwendigen Korrekturen durchzuführen.

[...] [E]in [Ergebnis], das im Zustand fertig gestellt ist [kann] auch durch andere Ereignisse, die nicht durch die Qualitätssicherung hervorgerufen wurden, wieder in den Zustand in Bearbeitung übergehen. So werden Ergebnisse beispielsweise [...] in nachfolgenden Fertigstellungsstufen des Projektes überarbeitet und damit in den Zustand in Bearbeitung versetzt.“

Abbildung 3.18 Allgemeine Vorgaben im VM_0 als Instanz von PBS_0

3.3.5 Projektablauf

Abbildung 3.19 zeigt die Entscheidungspunkte von VM_0. Das QS-Handbuch und der Projektplan sind initiale Produkte, die zu mehreren Entscheidungspunkten zugeordnet sind. Im Hinblick auf die querschnittliche Vorgabe aus Abbildung 3.18 sind diese jeweils nur neu zu überarbeiten, während für die Projektfortschrittsentscheidung, die ebenfalls zu mehreren Entscheidungspunkten zugeordnet ist, stets ein neues Ergebnis vorliegen muss. Insgesamt sind nicht alle Produkte explizit zu einem Entscheidungspunkt zugeordnet. Konkret betrifft dies die Prüfspezifikation, die Prüfprozedur und das Prüfprotokoll. Deren Fertigstellungszeitpunkt ergibt sich implizit aus der querschnittlichen Vorgabe aus Abbildung 3.18: diese müssen zum selben Meilenstein fertig gestellt werden, wie das geprüfte Ergebnis, auf das sie sich beziehen.

Abbildung 3.19 Entscheidungspunkte im VM_0 als Instanz von PBS_0

Die Reihenfolge, in der die Entscheidungspunkte zu durchlaufen sind, wird in Abbildung 3.20 dargestellt. Die beiden Projektdurchführungsstrategien unterscheiden sich lediglich darin, ob die vier Entscheidungspunkte zur Systemerstellung durchlaufen werden.

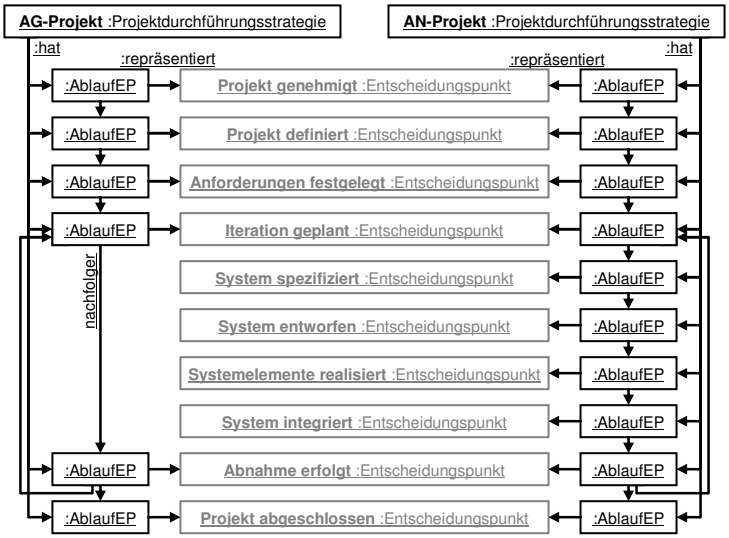


Abbildung 3.20 Projektdurchführungsstrategien im VM_0 als Instanz von PBS_0

4 Problemstellung

Das Projektcontrolling wird in der DIN 69901 [55] in abstrakter Weise definiert als die „*Sicherung des Erreichens der Projektziele durch: Soll-Ist-Vergleich, Feststellung der Abweichungen, Bewerten der Konsequenzen und Vorschlagen von Korrekturmaßnahmen, Mitwirkung bei der Maßnahmenplanung und Kontrolle der Durchführung*“. Unter der Operationalisierung des Projektcontrollings wird in dieser Arbeit die Unterlegung dieses abstrakten Projektcontrolling-Begriffs mit konkret und automatisiert ausführbaren Operationen verstanden. Zunächst zeigt Kapitel 4.1 anhand von Beispielen, dass und warum ein manuell durchzuführendes Projektcontrolling für den Projektleiter eine Herausforderung darstellt, und daher einer Automatisierung bzw. Unterstützung bedarf. Daraus wird in Kapitel 4.2 der in dieser Arbeit betrachtete Teil des Projektcontrollings definiert, und schließlich, in Kapitel 4.3, die zu lösende Aufgabenstellung – die zu automatisierenden bzw. durch Teilautomatisierungen zu unterstützenden manuellen Tätigkeiten – festgelegt.

4.1 Beispiele zur Schwierigkeit des manuellen Projektcontrollings

Ein Projekcontrolling kann nur so gut sein, wie es die betrachteten Eingaben zulassen. Beispielsweise hat ein Soll-Ist-Vergleich nur dann eine belastbare Aussage, wenn das aufgestellte Soll und das dazu ermittelte Ist detailliert genug sind. Dazu müssen im Projekt zu erstellende Dokumente (allgemein Ergebnisse) nicht nur als solche, sondern auch bezüglich ihrer Inhalte, Zustände und Versionen einschließlich deren Vernetzung betrachtet werden. Eine manuelle Betrachtung führt zu einem hohen Zeitaufwand sowie zu einer hohen Fehleranfälligkeit bei der Aktualisierung des Solls und des Soll-Ist-Vergleichs – besonders dann, wenn das Ist (projektbedingt) häufigen Änderungen unterliegt und (bedingt durch die heutigen Systeme) sehr umfangreich ist.

Vorgehensmodelle geben einen Rahmen für die Projektdurchführung und damit auch einen Rahmen für das Projektcontrolling. Deren Vorgaben müssen jedoch, da als für viele Projekte anwendbar angelegt, für das konkret durchgeführte Projekt interpretiert bzw. ausgestaltet werden. Die Vorgaben nutzen dabei Quantifizierungen, Bedingungen und Bezüge zu verschiedenen Konzeptebenen. Das Anwenden ist somit zeitlich aufwendiger und Fehleranfälliger als etwa das Abarbeiten von Checklisten. In den nachfolgenden Unterkapiteln wird dies anhand von konkreten Beispielen verdeutlicht.

4.1.1 Berücksichtigung von Dokumenten und Dokumentinhalten

Eine Vielzahl von Vorgaben basieren nicht nur auf der Auswertung vorhandener Dokumente, sondern auch auf der Auswertung derer Inhalte. Als Beispiel sei folgende Vorgabe aus VM_0 betrachtet:

> „Für jeden spezifizierten Prüffall wird eine Prüfprozedur […] erstellt.“

Für die Befolgung dieser Vorgabe muss der Projektleiter bei der Erstellung (und jeder Änderung einer Prüfspezifikation) für jeden darin aufgeführten Prüffall prüfen, ob eine entsprechende Prüfprozedur im Projektplan einplant wurde. Zusätzlich hat er auch zu prüfen, ob umgekehrt jede im Projektplan aufgeführte Prüfprozedur in einer der Prüfspezifikationen noch vorhanden ist. Sofern im Projektplan nicht für jeden Prüffall angegeben ist, durch welche Prüfspezifikation es erzeugt wurde, müsste der Projektleiter bei Änderung einer Prüfspezifikation alle Prüffälle mit allen Prüfspezifikationen abgleichen. Alternativ könnte er einen Vergleich zur Vorversion durchführen, woraus sich die aus dem Projektplan zu entfernenden Prüffälle ebenfalls ergeben würden. Aber auch das bedeutet einen entsprechenden Aufwand und erfordert die Disziplin, dies bei jeder Änderung tatsächlich zu

tun: Bei Auslassung auch nur einer Änderung kann nicht mehr festgestellt werden aus welchem Dokument ein Prüffall entfernt wurde.

4.1.2 Berücksichtigen von Zuständen und Versionen

Einige Vorgaben steuern die Erarbeitung von Ergebnissen durch Definition von zu erreichenden Zwischenzuständen. Bestimmte Zustände können dabei Eingabe bzw. Betrachtungsgegenstand weiterer Ergebnisse sein, so dass nicht nur der aktuellste Stand eines Dokuments, sondern auch dessen Versionshistorie zu berücksichtigen ist. Als Beispiel sei folgende Vorgabe aus VM_0 betrachtet:

> „[…] Ist eine eigenständige Qualitätssicherung notwendig, so wechselt das [Ergebnis] zuerst in den Zustand vorgelegt und nach der erfolgreichen Prüfung in den Zustand fertig gestellt. Andernfalls geht das [Ergebnis] sofort nach der erfolgreichen Eigenprüfung in den Zustand fertig gestellt über. Ist eine Prüfung nicht erfolgreich, so muss das [Ergebnis] entsprechend überarbeitet und erneut qualitätsgesichert werden. Sind dabei relevante Produktabhängigkeiten verletzt, so sind die Produktverantwortlichen der beteiligten [Ergebnisse] für die Beseitigung der Inkonsistenz verantwortlich. Dies kann auch dazu führen, dass die [V]erantwortlichen [...] entscheiden, dass ein bereits fertig gestelltes [Ergebnis] wieder in den Zustand in Bearbeitung gesetzt wird, um die notwendigen Korrekturen durchzuführen."

Für die Einplanung neu zu erstellender Versionen muss der Projektleiter

- für jedes Dokument oder dessen Art klären, welcher Zustandsübergang überhaupt vorzunehmen ist. Dazu muss er im QS-Handbuch nachsehen – oder dies im Gedächtnis behalten.
- bei bestimmten Zustandsübergängen neue Dokumente oder neue Versionen existierender Dokumente einplanen – hier konkret für neue Prüfprotokolle.
- bei negativen Prüfbefunden für die davon betroffenen Dokumente neue Versionen zu deren Überarbeitung einplanen. Sofern die Verantwortlichen jener Produkte die Überarbeitung bereits in Eigenregie begonnen haben, muss der Projektleiter die dann entsprechenden bereits erstellten Versionen mit seinen Einplanungen synchronisieren.

4.1.3 Berücksichtigung des relevanten Vorgehensmodellausschnitts

Vorgehensmodelle decken eine große Bandbreite an möglichen Projekten mit dazu passenden Vorgaben ab. Für eine konkrete Organisation, Abteilung oder ein konkretes Projekt ist in aller Regel nur ein Teil davon relevant. Um diesen Teil zu bestimmen, enthalten einige Vorgehensmodelle (vgl. Kapitel 3) Vorgaben zu ihrer eigenen Anpassung. Diese Vorgaben können sowohl vor einem Projekt ausgeführt werden (wodurch sie aus Sicht eines Projektes statisch sind), oder auch während des Projektes (wodurch sie aus Sicht eines Projektes dynamisch sind). Letzterer Fall stellt eine höhere Belastung für den Projektleiter dar, da jene Vorgaben sich auf die im Projekt erarbeiteten Ergebnisse und deren Inhalte beziehen. Als Beispiel sei dazu folgende Vorgabe aus VM_0 betrachtet:

> „[…] Wurde im Produkt Systemarchitektur mindestens eine SW-Einheit identifiziert, muss im Projekthandbuch der Vorgehensbaustein SW-Entwicklung ausgewählt werden. […]"

Ein Projektleiter muss zu Befolgung dieser Vorgabe das Vorgehensmodell bereits gut kennen, um zu wissen, wann und wie solche Vorgaben anzuwenden sind. Dies stellt vor allem für Neueinsteiger ein großes Hindernis dar, wodurch die Vorteile eines Vorgehensmodells nicht in vollem Umfang genutzt werden können.

4.2 Problemdomäne: Projektcontrolling

In diesem Kapitel wird der betrachtete Teil des Projektcontrollings im Rahmen der Projektdurchführung konkret, aber unabhängig[29] von einem bestimmten Vorgehensmodell, definiert.

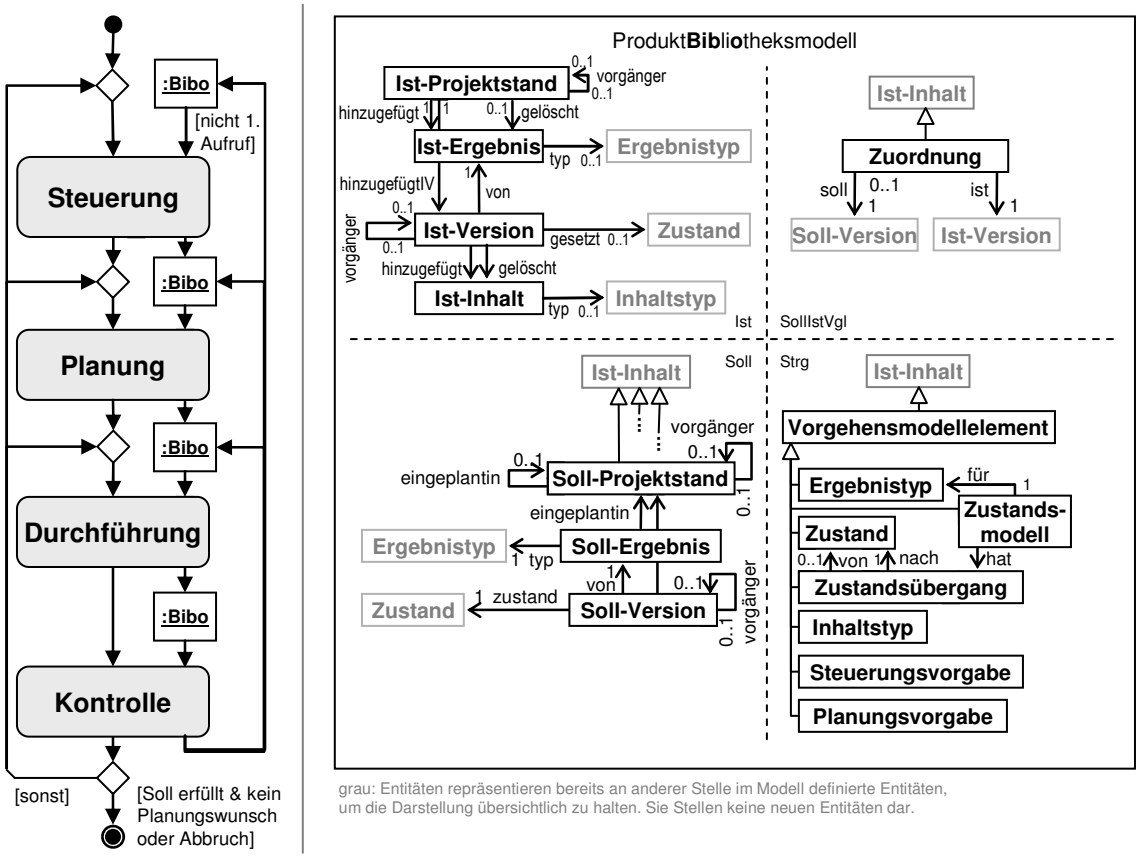


Abbildung 4.1 Problemdomäne

Projektcontrolling. Abbildung 4.1 zeigt das in dieser Arbeit genutzte Verständnis des Projektcontrollings: Das Projektcontrolling ist ein Regelkreis mit den Schritten Steuerung, Planung, Durchführung und Kontrolle, die sich laufend abwechseln[30]. Ein Projekt beginnt mit der Steuerung, in dem das für das Projekt anzuwendende Vorgehensmodell definiert wird. Diese Definition muss nicht abschließend sein, sondern kann für weitere Steuerungsschritte anpass- bzw. ausgestaltbar gelassen werden. Auch ist es freigestellt, anstelle eines neu definierten Vorgehensmodells ein bereits existierendes, ggf. in einer angepassten Form, zu verwenden. Beispielsweise kann dies VM_0 sein. In jedem Falle bildet das definierte Vorgehensmodell die Grundlage für die Planung, bei der anhand der Planungsvorgaben des Vorgehensmodells das Soll aufgestellt wird. Das Soll wiederum ist Eingabe für die Durchführung und dient dem Projektteam, zusammen mit allgemeinen Vorgaben des Vorgehensmodells (z.B. Beschreibungen von Ergebnistypen), als Anleitung zur Erstellung bzw. Überarbeitung des Ist. Bei der Kontrolle wird das Ist dem Soll durch Zuordnung von Ergebnissen (genauer: deren Versionen) gegenübergestellt. Je nach Erfüllungsgrad und Stand

[29] Dies schließt nicht aus, dass zur Erläuterung der Festlegungen auf Beispiele zurückgegriffen wird.

[30] In einem Schritt können beliebig viele, also auch keine, Handlungen vorgenommen werden. Bei häufigen Schrittwechseln entsteht so eine Scheinparallelität. Für echte Parallelität sind Merging-Techniken zu nutzen (siehe z.B. [87]). Eine vereinfachte Variante wäre z.B. ein Locking auf z.B. der Dokumentebene, so dass in dem Fall das Merging trivial ist – zumindest was die Parallelität des Vornehmens der Änderungen (also das Editieren) anbelangt. Erkennung und Behandlung von Konflikte wären dann nach wie vor notwendig, sind in dieser Arbeit aber nicht im Fokus.

des Ist entscheidet der Projektleiter, ob mit der Durchführung fortzufahren, die Planung zu überarbeiten oder ob sogar die eingesetzten Vorgehensmodellvorgaben zu verändern sind. Ist das Soll erfüllt, und besteht kein weiterer Wunsch das Soll zu verändern, ist das Projekt beendet. Das Projektende kann auch durch einen Abbruch ausgelöst werden. Ein Schnappschuss des Projektes wird mit einer *Produktbibliothek* beschrieben (Abbildung 4.1, rechts). Die Details zur Produktbibliothek (Strukturen) und den Schritten (Verhalten) werden in den Unterkapiteln 4.2.1 bis 4.2.5 behandelt.

4.2.1 Produktbibliothek

Die Produktbibliothek gliedert die Informationen eines Projektes nach den vier Teilmodellen

- Strg – für den Schritt Steuerung,
- Soll – für den Schritt Planung,
- Ist – für den Schritt Durchführung und
- SollIstVgl – für den Schritt Kontrolle.

Im Folgenden werden zunächst die Konstrukte der Teilmodelle entsprechend dem Regelkreis des Projektcontrollings erklärt und abschließend formal definiert. Der Eingängigkeit halber wird der Durchlauf mit dem Planungsschritt begonnen.

Ist-Projektstand, -Ergebnis, -Version und -Inhalt. Das in einem Projekt erarbeitete Ist gliedert sich in Ergebnisse und Inhalte. Ein Ergebnis ist in der Realität mit einem sich über die Zeit verändernden Dokument[31] gleichzusetzen. Es kann also nicht als genau ein Dokument verstanden werden, sondern als die Menge dessen Schnappschüsse. Die Veränderung des Dokuments erfolgt auf der Ebene von Inhalten. Ein Inhalt kann dabei von Anfang an über die gesamte Zeit vorhanden sein (wodurch er in jedem Schnappschuss zu sehen ist) oder erst später erstellt bzw. gelöscht werden (wodurch er nur in einigen Schnappschüssen auftaucht). In der Produktbibliothek werden Schnappschüsse nicht direkt, sondern als Differenz je zwei aufeinanderfolgender Schnappschüsse abgelegt. Eine solche Differenz wird als Version bezeichnet. Die Assoziation vorgänger bildet eine totale Ordnung über den Versionen. Die Assoziationen hinzugefügt und gelöscht beschreiben welche Inhalte in einer Version hinzugekommen sind oder entfernt wurden. Um die Inhalte eines Schnappschusses zu einer Version zu ermitteln, müssen die Hinzufügungen und Löschungen aller Vorgängerversionen aufsummiert werden. Ein Inhalt kann beliebige Informationen tragen. Insbesondere kann ein Inhalt weiter strukturiert sein, ohne dass die Struktur aus der Perspektive des Projektcontrollings bekannt ist. Beispielsweise kann ein Inhalt ein Stück Java-Programmcode sein der somit gemäß der Sprache Java strukturiert ist (in Klassen, Funktionen, Anweisungen usw.). Diese Struktur ist jedoch nicht für die allgemeine Problemdomäne relevant.

Durch die Assoziation gesetzt kann mit jeder Version ein (neuer) Zustand gesetzt werden. Der aktuelle Zustand eines Ist-Ergebnisses entspricht dem Zustand der bezüglich der Vorgänger-Relation als letztes gesetzt wurde.

Über Ist-Projektstände wird verfolgt, wann Ist-Ergebnisse erstellt, um neue Versionen erweitert, oder wieder gelöscht wurden (bspw. dann, wenn sich diese als irrelevant ergeben haben). Eine Löschung wird jedoch nicht durch ein tatsächliches Entfernen aus einer Produktbibliothek festgeschrieben – sondern virtuell über die Assoziation gelöscht. Ist-Ergebnisse können nur gelöscht werden, wenn sie in einem vorherigen Projektstand hinzugefügt wurden. Ist-Ergebnisse können nur einmalig hinzugefügt oder gelöscht werden. Ein einmal gelöschtes Ist-Ergebnis kann später nur als Kopie (mit einer neuen Id) wieder

[31] Anstelle des Dokumentes kann auch, allgemeiner, eine Datei eingesetzt werden. Sofern auch Hardware erstellt wird, kann es sich auch um ein entsprechendes Werkstück handeln.

hinzugefügt werden. Versionen können nicht gelöscht werden. Stattdessen kann eine ältere Version kopiert und diese Kopie als die neueste Version gesetzt werden, wodurch zwischenzeitliche Änderungen effektiv rückgängig gemacht werden. In einem Ist-Projektstand können mehrere Ist-Versionen, sowohl zu unterschiedlichen, als auch zum selben Ist-Ergebnis, angelegt werden. Analog können in einem Ist-Projektstand mehrere Ist-Ergebnisse erstellt oder gelöscht werden.

Soll-Projektstand, -Ergebnis und -Version. Ein Soll-Element (Soll-Projektstand, -Ergebnis, -Version) ist eine Vorgabe an das Projektteam, ein dazu passendes Ist-Element (Ist-Projektstand, -Ergebnis, -Version) im Rahmen der Durchführung zu erstellen. Durch die Assoziationen eingeplantin und vorgänger wird eine Reihenfolge vorgegeben, in der die Ergebnisse erstellt sein müssen. Nicht festgelegt wird jedoch, wann mit der Arbeit an den jeweiligen Ergebnissen begonnen werden darf. So kann die Reihenfolge beispielsweise als Arbeitsbeginnreihenfolge oder als Fertigstellungsreihenfolge (wie im V-Modell XT) verstanden werden. Ein Soll-Ergebnis fordert die Erarbeitung, eine Soll-Version die Überarbeitung eines Ist-Ergebnisses (mit Ausnahme der ersten). Soll-Projektstände können hierarchisch sein. Die Hierarchie wird dadurch ausgedrückt, dass ein Soll-Projektstand in einem anderen eingeplant ist. Vorgängerbeziehungen sind entsprechend nur zwischen solchen Soll-Projektständen erlaubt, die entweder zum selben Soll-Projektstand oder gar keinem (oberste Hierarchiestufe) eingeplant sind.

Zuordnung. Auf der Ebene von Versionen wird durch explizite Versionszuordnungen ausgedrückt, welche Ist-Version welche Soll-Version erfüllt. Die Zuordnung eines Ist-Ergebnisses zu einem Soll-Ergebnis ergibt sich implizit über die Versionszuordnung dessen Ist-Versionen. Zur Eindeutigkeit müssen entsprechend über die Planungsvorgaben zu jedem Soll-Ergebnis auch mindestens eine Soll-Version eingeplant und erstellt werden, und Ist-Versionen dürfen nur zum selben Soll-Ergebnis zugeordnet werden. Die Ebene der Projektstände ist für diese Arbeit nicht[32] interessant und wird daher nicht explizit betrachtet.

Vorgehensmodellelement. Die Vorgehensmodellelemente gliedern sich in Planungs- und Steuerungsvorgaben einerseits und in die restlichen Elemente, im Folgenden als Hilfselemente bezeichnet, andererseits. Hilfselemente vereinfachen die Definition von Planungs- und Strukturvorgaben: Beispielsweise kann in einer Planungsvorgabe ein Element der Produktbibliothek kurz anhand des Typs charakterisiert werden – anstelle einer umfangreichen Beschreibung der ggf. umgebenden Struktur (vgl. Motivation zur Einführung von Typen für Graphen in Kapitel 2.4.5). Neben Typen können auch weitere Elemente (wie z.B. Themen oder Entscheidungspunkte) eingeführt werden, um komplexere Planungs- und Steuerungsvorgaben zu definieren.

Zustände sind ein Mechanismus zur Typung von Versionen. Darauf aufbauend können Zustandsübergänge definiert werden, die ihrerseits, in Kombination mit Planungsvorgaben, den Ablauf einer Überarbeitung eines Ergebnisses steuern können. Dazu bildet zunächst ein Zustandsmodell die Grundlage, welches angibt, welche Zustände und Zustandsübergänge für welche Ergebnistypen gelten (Assoziation giltfür). Die Menge aller Zustände für einen Ergebnistyp ergibt sich aus den zum Zustandsmodell des Ergebnistyps durch die Assoziation hat zugeordnete Zustandsübergänge. So können Zustände und Zustandsübergänge auch von mehreren Zustandsmodellen wiederverwendet werden. Startzustände werden nicht explizit, sondern über einen Start-Zustandsübergang angegeben, der sich durch Abwesenheit eines vorherigen Zustands (Assoziation von ist nicht belegt) auszeichnet. So können

[32] Soll-Projektstände selbst, also Knoten des Typs Soll-Projektstand ohne der dazu eingeplanten Soll-Ergebnisse und –Versionen, sind in ihrer Anzahl gering und unterliegen deutlich seltener Planungsänderungen.

Zustandsmodelle behandelt werden die verschiedene, mit Bedingungen versehene Startzustände haben.

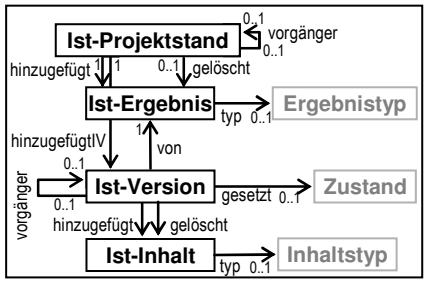
Planungsvorgabe. Eine Planungsvorgabe beschreibt wie anhand der aktuellen Produktbibliothek Soll-Elemente optional oder verpflichtend zu erstellen sind.

Steuerungsvorgabe. Steuerungsvorgaben definieren ob und wie Vorgehensmodellelemente anzuwenden bzw. auszugestalten sind. Im einfachsten Fall definiert eine Steuerungsvorgabe, dass beispielsweise ein konkreter Ergebnistyp im aktuellen Projekt relevant ist. Die steuernde Wirkung dieser Vorgabe tritt ein, wenn Planungsregeln auf Steuerungsvorgaben eingehen. Dies ist der Fall wenn beispielsweise eine Planungsvorgabe das Erstellen eines Soll-Elements für jeden relevanten Ergebnistyp fordert. Steuerungsvorgaben können selbst Gegenstand anderer Steuerungsvorgaben sein. Beispielsweise kann eine Steuerungsvorgabe vorgeben, welche Kombinationen von einfachen Steuerungsvorgaben zur Festlegung der relevanten Ergebnistypen erlaubt sind. Für ein noch konkretes Beispiel, kann das Tailoring aus VM_0 auf diese Weise beschrieben werden. Darüber hinaus können Steuerungsvorgaben auch die Veränderung von existierenden Steuerungsvorgaben vorgeben, z.B. in Form von Änderungsoperationen (vgl. [88]).

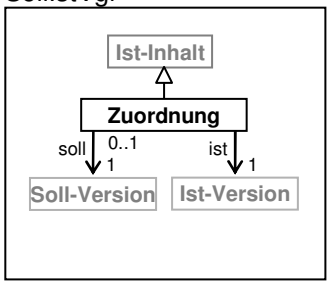
Definition 4.1: Produktbibliotheksmodell
*Das **Produktbibliotheksmodell** ist ein Modell aus* MODEL*, wird mit* Bibomodell *notiert, und ist wie folgt definiert:*

Bibomodell $=_{def}$ Strg + Soll + Ist + SollIstVgl
wobei

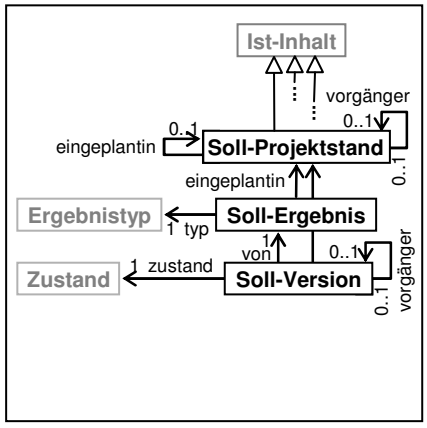
Ist=



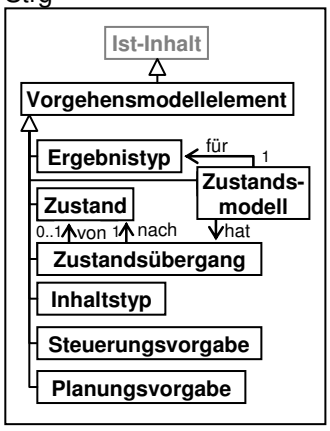
SollIstVgl=



Soll=



Strg=



Die vier Teilmodelle überschneiden sich in einigen Konstrukten. Durch die Addition der vier Graphen entsteht jedoch das in Abbildung 4.1 gezeigte Produktbibliotheksmodell.

Definition 4.2: Initialproduktbibliothek, Teilbibliothek
Die ***Initialproduktbibliothek*** Initialbibo ∈ GRAPH*, mit* Initialproduktbibliothek *notiert, ist wie folgt definiert:*

Initialbibo $=_{def}$

Die Ids BiboStrg, BiboSoll, BiboSollIstVgl ∈ GE *werden als* ***Teilbibliotheken*** *bezeichnet.*

Die Initialproduktbibliothek enthält einen Ist-Projektstand (mit der reservierten Id initialstand ∈ GE), der wiederum ein Ist-Ergebnis für jedes Teilmodell (außer Ist) enthält. In diesen werden die jeweiligen Informationen versioniert aber auch dediziert abgelegt. Zur Unterscheidung der Ist-Ergebnisse werden die drei Ids BiboStrg, BiboSoll, BiboSollIstVgl ∈ GE reserviert und verwendet. Neben den Ids werden im Foglenden auch die Inhalte der entsprechenden Ist-Ergebnisse als Teilbibliotheken bezeichnet.

Definition 4.3: Produktbibliothek
Die Menge BIBO *aller* ***Produktbibliotheken*** *ist wie folgt definiert:*

$$BIBO =_{def} \{ g \in PRJ \mid g \in Ext(\text{Bibomodell}), \text{Initialbibo} \subset g \}$$

Definition 4.4: Vorgänger einer Ist-Version
Sei eine Produktbibliothek B ∈ BIBO *gegeben. Die* ***Vorgängerfunktion für Ist-Versionen*** *in* B *sei mit* $vorg_{IV}(B)$ *notiert und wie folgt definiert:*

$$vorg_{IV}(B) = \{ (v,vv) \in instances(\text{Ist-Version},B)^2 \mid vv \in collect(v,\text{vorgänger},B) \}$$

Der Vorgänger einer Ist-Version v ist diejenige Ist-Version, die von v durch einen Link vom Typ vorgänger referenziert wird. Für die „erste“ Ist-Version ist der Vorgänger nicht definiert.

Definition 4.5: Vorgänger einer Soll-Version
Gegeben seien eine Produktbibliothek B ∈ BIBO *und zwei Soll-Versionen v, vv* ∈ instances(Soll-Version,B). *Das Prädikat* $istvorg_{sv}$ *gibt an ob die Version* vv *die direkte Vorgängerversion von* v *in* B *ist, und ist wie folgt definiert:*

$$istvorg_{SV}\langle v,vv,B\rangle =_{def} vv \in collect(v,\text{vorgänger},B) \text{ oder}$$
$$collectors(vv,\text{vorgänger},B)=\emptyset \wedge$$
$$collect(v,\text{vorgänger},B)=\emptyset \wedge$$
$$istvorg_{SV}\langle any(collect(vv,\text{eingeplantin},B)),any(collect(v,\text{eingeplantin},B)),B\rangle$$

Die ***Vorgängerfunktion für Soll-Versionen*** *in* B *sei mit* $vorg_{SV}(B)$ *notiert und wie folgt definiert:*

$$vorg_{SV}(B) = \{ (v,vv) \in instances(\text{Soll-Version},B)^2 \mid istvorg_{SV}\langle v,vv,B\rangle \}$$

Eine Soll-Version vv ist genau dann ein direkte Vorgängerversion von v, wenn es einen Link von v nach vv mit dem Typ vorgänger gibt, oder falls es auf einer höheren Planungsebene (Soll-Projektstände) ein Äquivalent für einen solchen Link gibt. Im einfachsten Fall trifft dies zu wenn vv die letzte, und v die erste Version in ihren jeweiligen Soll-Projektständen sind, und deren Soll-Projektstände direkt über einen Link vom Typ vorgänger verbunden sind. Die komplexeren Fälle erlauben es, dass die Soll-Projektstände ihrerseits nicht direkt, sondern auf analoge Weise über eine noch höhere Planungsebene verbunden sind.

Definition 4.6: Inhalte einer Ist-Version
Sind eine Produktbibliothek B ∈ BIBO *und ein* v ∈ instances(Ist-Version,B) *gegeben, so sind die* ***Inhalte*** *von* v *bezüglich* B*, notiert als* $inhalte_{IV}(v,B)$*, wie folgt definiert:*

$inhalte_{IV}(v, B) =_{def}$ falls vv n.def. dann $\emptyset$ sonst ($inhalte_{IV}(vv, B) \cup$ added) / deleted
wobei
vv = $[vorg_V(B)](v)$
added = collect(v,hinzugefügt,B)
deleted = collect(v,gelöscht,B)

Die Inhalte einer Ist-Version ergeben sich durch das chronologische Aufrechnen aller Hinzufügungen und Löschungen, beginnend mit der ersten, und endend mit der betrachteten Version v.

Definition 4.7: Zustand einer Version
Sind eine Produktbibliothek B ∈ BIBO *und ein* v1 ∈ instances(Ist-Version,B) *(bzw. ein* v2 ∈ instances(Soll-Version,B)) *gegeben, so ist der* ***Zustand*** *von* v1 *(bzw. von* v2*) bezüglich* B*, notiert als* $zustand_{IV}(v1,B)$ *(bzw. als* $zustand_{SV}(v2,B)$ *), wie folgt definiert:*

$zustand_{IV}(v1,B) =_{def}$ falls |G|=1 dann any(G) sonst $zustand(vorg_{IV}(v1),B)$ mit
G = collect(v1,gesetzt,B)

$zustand_{SV}(v2,B) =_{def}$ any(collect(v2,zustand,B))

Für eine Ist-Version ergibt sich der Zustand ist der über gesetzt referenzierte, sofern ein Zustand überhaupt referenziert wird. Falls das nicht der Fall ist, ergibt sich der Zustand aus der vorherigen Ist-Version. Im Extremfalll ist der Zustand nicht definiert. Für eine Soll-Version ist der Zustand derjenige, der über zustand referenziert wird.

Definition 4.8: Versionen eines Ergebnisses
Sind eine Produktbibliothek B ∈ BIBO *und ein* e ∈ instances(y,B) *mit* y ∈ {Soll-Ergebnis, Ist-Ergebnis} *gegeben, so ist die Menge der* ***Versionen*** *von* e *bezüglich* B*, notiert als* versionen(v,B)*, wie folgt definiert:*

versionen(e,B) $=_{def}$ collectors(e,von,B) ∩ (instances(Ist-Version,B)∪instances(Soll-Version,B))

Definition 4.9: Aktuellste Version eines Ist-Ergebnisses
Sind eine Produktbibliothek B ∈ BIBO *und ein* e ∈ instances(Ist-Ergebnis,B) *gegeben, so ist die* ***aktuellste Version*** *von* e *bezüglich* B*, notiert als* $aktversion_{IE}(v,B)$*, wie folgt definiert:*

$aktversion_{IE}(e,B) =_{def}$ any(versionen(e,B) / $WB(vorg_{IV}(B))$)

Die aktuellste Version ist diejenige, die zu e gehört und kein Vorgänger einer anderen Version ist.

Definition 4.10: Vorgänger eines Ist-Projektstandes
Sei eine Produktbibliothek B ∈ BIBO *gegeben. Die* ***Vorgängerfunktion für Ist-Projektstände*** *in* B *sei mit* $vorg_{Prjstd}(B)$ *notiert und wie folgt definiert:*

$$vorg_{Prjstd}(B) = \{ (p,pv) \in instances(Ist\text{-}Projektstand,B)^2 \mid pv \in collect(p,vorgänger,B) \}$$

Der Vorgänger eines Ist-Projektstandes p ist derjenige, der von p durch einen Link vom Typ vorgänger referenziert wird. Für den ersten Ist-Projektstand ist der Vorgänger nicht definiert.

Definition 4.11: Inhalte eines Ist-Projektstandes
Sind eine Produktbibliothek B ∈ BIBO *und ein* p ∈ instances(Ist-Projektstand,B) *gegeben, so sind die* ***enthaltenen Ist-Ergebnisse*** *von* p *bezüglich* B*, notiert als* $inhalte_{Prjstd,IE}(p,B)$*, wie folgt definiert:*

$inhalte_{Prjstd,IE}(p, B) =_{def}$ falls pv n.def. dann $\emptyset$ sonst $(inhalte_{Prjstd,IE}(pv , B) \cup added) / deleted$
wobei
pv = $[vorg_{Prjstd}(B)](p)$
added = collect(p,hinzugefügt,B)
deleted= collect(p,gelöscht,B)

Die ***enthaltenen Ist-Versionen*** *von* p *bezüglich* B*, notiert als* $inhalte_{Prjstd,IV}(p,B)$*, sind wie folgt definiert:*

$inhalte_{Prjstd,IV}(p, B) =_{def}$ falls pv n.def. dann $\emptyset$ sonst $(inhalte_{Prjstd,IV}(pv , B) \cup added) / deleted$
wobei
pv = $[vorg_{Prjstd}(B)](p)$
added = collect(p,hinzugefügtIV,B)
deleted = $\bigcup$ ie ∈ collect(p,gelöscht,B) : versionen(ie)

Die Inhalte eines Ist-Projektstandes sind eine Menge von Ergebnissen (bzw. Versionen). Diese werden gemäß der Hinzufügungen und Löschungen chronologisch aufgerechnet, beginnend mit dem aller ersten Ist-Projektstand, und endend mit dem betrachteten Ist-Projektstand p. Wird bei der Aufrechnung ein Ist-Ergebnis gelöscht, so führt dies auch zur Löschung aller dessen Ist-Versionen. Da die Ist-Versionen der Ist-Ergebnisse über Ist-Inhalte verfügen, enthält ein Ist-Projektstand implizit auch Ist-Inhalte.

Definition 4.12: Aktuellster Ist-Projektstand
Ist eine Produktbibliothek B ∈ BIBO *gegeben, so ist der* ***aktuellste Projektstand*** *bezüglich* B*, notiert als* aktprjstd(B)*, wie folgt definiert:*

$$aktprjstd(B) =_{def} any(instances(Ist\text{-}Projektstand,B) / WB(vorg_{Prjstd}(B))$$

Der aktuellste Ist-Projektstand ist diejenige, der kein Vorgänger eines anderen Ist-Projektstandes ist.

Definition 4.13: Schrittfunktion
Gegeben sei eine (referenziell nicht transparente) Funktion f : BIBO → BIBO*, sowie ein Modell* m ∈ {Strg,Soll,Ist,SollIstVgl}*. Wenn* f *jede Eingabe nur bezüglich des Modells* m *erweitert, wird* f *als eine* ***Schrittfunktion*** *für* m *bezeichnet. Die Erweiterung jeder Eingabe bezüglich eines Modells* m *wird mit* schrittfunktion<f,m> *notiert und ist wie folgt definiert.*

$$schrittfunktion<f,m> =_{def} [\forall B \in BIBO, \text{für alle Aufrufe: } [B^- = \emptyset] \wedge [B^+ \subseteq (N \cup E)]$$

wobei
B' = f(B), einmalig ausgeführt
$B^- = B/B'$
$B^+ = B'/B$

N = instances(Ist-Projektstand) ∪
[falls m=Ist dann
$[\bigcup$ n ∈ instances(Ist-Ergebnis)/WB(teilbibo) : {n} ∪
$[\bigcup$ v ∈ versionen(n,B') : {v} ∪ $inhalte_{IV}$(v,B')]]
sonst
$[\bigcup$ v ∈ versionen(teilbibo(m),B') : {v} ∪ $inhalte_{IV}$(v,B')]]
E = [$\bigcup$ e∈edges(B') mit $\underline{s}_{B'}$(e) ∈ N : { e }]

teilbibo = { Strg ↦ BiboStrg, Soll ↦ BiboSoll, SollIstVgl ↦ BiboSollIstVgl }

Eine Funktion f : BIBO → BIBO *mit* schrittfunktion<f,m> *wird auch mit* f : BIBO $\rightarrow_m$ BIBO *notiert.*

Eine Schrittfunktion für ein Modell m repräsentiert das Verhalten eines Projektmitglieds beim ausführen des entsprechenden Schrittes im Rahmen des Projektcontrollings: Eine Schrittfunktion für das Modell Soll repräsentiert beispielsweise den Projektleiter beim Ausführen des Planungsschritts; eine Funktion für das Modell Ist das Projektteam beim Ausführen des Durchführungsschritts. Da das Verhalten von Personen nicht umfassend formal greifbar ist, wird eine Schrittfunktion allgemein als referenziell nicht transparent angenommen. Entsprechend wird in der Formalisierung die Ausführung eines Schritts zu B' nur einmalig ausgeführt. Die dabei gelöschten und neu erzeugten Elemente ergeben sich durch die beiden Differenzen aus B' und B. Da mit einer Produktbibliothek die Gesamte Projekthistorie erfasst wird, dürfen keine „tatsächlichen" Löschungen vorgenommen werden (B^- muss folglich leer sein). Stattdessen kann nur virtuell gelöscht werden – durch Verwendung des Konstrukts gelöscht als Referenz auf ein Ist-Ergebnis oder ein Ist-Inhalt. Die neu hinzugefügten Graphelemente (erfasst durch B^+) müssen zu dem jeweils ausgeführten und durch m bestimmten Schritt „passen". Dies ist der Fall, wenn die Knoten und Kanten aus B^+ in N bzw. E liegen. Für einen Durchführungsschritt (m=Ist) umfasst N alle Ist-Projektstände sowie alle Ist-Ergebnisse (außer denen für die Teilmodelle Soll, Ist und SollIstVgl) einschließlich deren Version und wiederum deren Inhalte. Für jeden anderen Schritt m umfasst N wiederum alle Ist-Projektstände sowie alle Versionen und Inhalte der zugehörigen Teilbibliothek teilbibo(m).

4.2.2 Steuerung

Unter einem Steuerungsschritt wird die Bearbeitung der Teilbibliothek Strg verstanden. Ausgehend von einem neuen Projekt das mit der Initialproduktbibliothek beginnt, zeigt Abbildung 4.2 ein beispielhaftes Ergebnis eines Steuerungsschrittes – in Form der resultierenden Produktbibliothek B1. Alle durch diesen Schritt gegenüber der Initialproduktbibliothek hinzugefügten Elemente sind grün gefärbt. Konkret ist dies ein neuer Ist-Projektstand mit einer neuen Ist-Version für die Teilbibliothek Strg, die wiederum eine Reihe von Vorgehensmodellelementen hinzufügt.

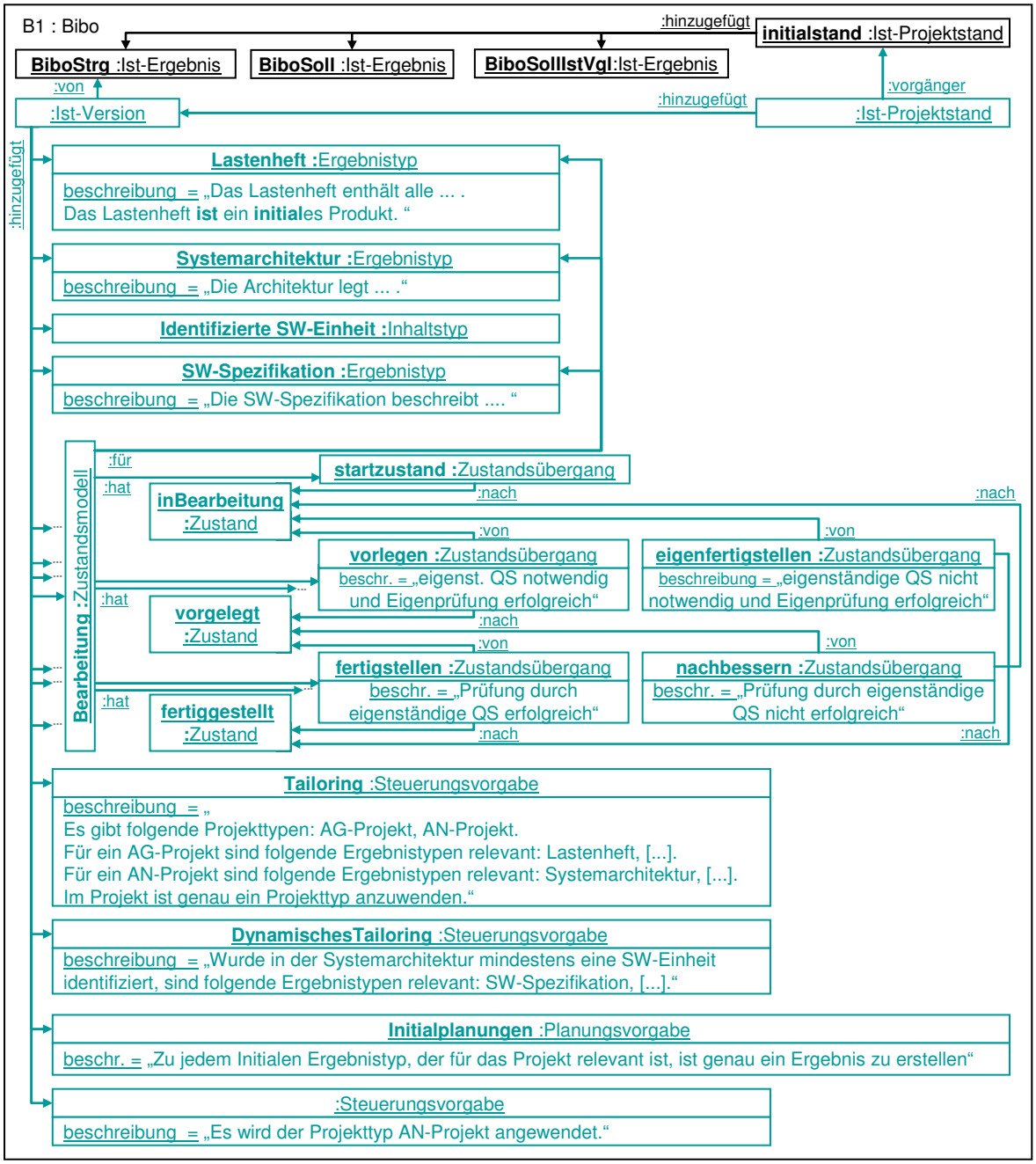


Abbildung 4.2 Beispiel für das Ergebnis eines Steuerungsschrittes

Sämtliche Vorgehensmodellelemente sind aus VM_0 entnommen. Die Steuerungsvorgaben müssen dabei nicht zwangsläufig alle zu einem Zeitpunkt festgelegt werden. In Abbildung 4.3 wird erläutert, wie die Situation in B1 auch durch mehrere (Teil-)Schritte erreichbar ist.

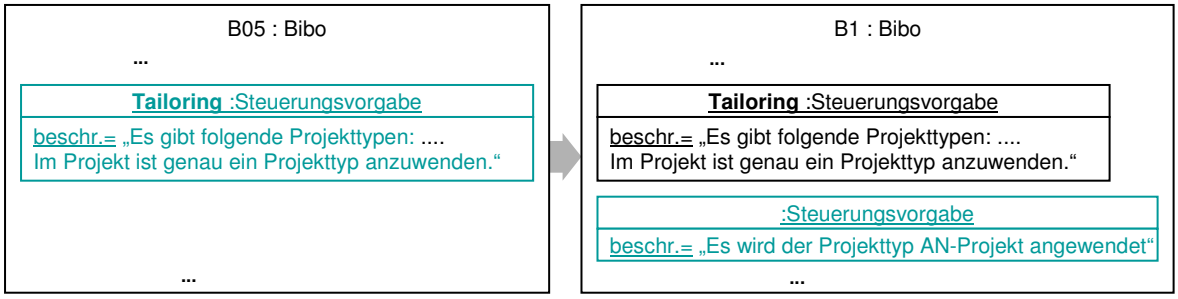


Abbildung 4.3 Nutzung der Organisationsstufen für stufenweise Steuerungsvorgaben

Mit B05 wurde zunächst die Steuerungsvorgabe Tailoring definiert – z.B. von einem Prozessingenieur und unabhängig von einem konkreten Projekt. Die Produktbibliothek B05 könnte also als eine allgemeine Ausgangsbasis für mehrere Projekte angelegt worden sein. Erst in einem konkreten Projekt wurde die Produktbibliothek B1 erstellt – diesemal von einem Projektleiter. Dieser wendete die Vorgaben des Prozessingeineurs auf sein konkretes Projekt an, und wählte den für ihn geeigneten Projekttyp aus. Die Auswahl hat ihrerseits eine steuernde Wirkung – in erster Linie an den Projektleiter selbst, und somit indirekt und im Allgemeinen auch an das restliche Projektteam. Entsprechend wird diese Auswahl in Form einer vom Projektleiter neu erstellten Steuerungsvorgabe zur Teilbibliothek Strg hinzugefügt.

Der Steuerungsschritt lässt sich als eine referenziell nicht transparante Funktion steuerungsschritt: BIBO $\rightarrow_{Strg}$ BIBO verstehen, wobei die Ausgabe den in ihr enthaltenen Steuerungsvorgaben nicht widersprechen darf. Als Eingabe wird die gesamte Produktbibliothek betrachtet, da Steuerungsvorgaben in Abhängigkeit des bisher Erzeugten formuliert werden können sollen (z.B. dynamisches Tailoring), wozu wiederum die Auswertung der „richtigen“ Ist-Elemente (Teilbibliothek Ist) notwendig ist. Dies wiederum kann jedoch ohne Betrachtung der Zuordnungen (Teilbibliothek SollIstVgl und folglich auch der Teilbibliothek Soll) nicht eindeutig erfolgen, da die heranzuziehenden Ist-Versionen sonst „geraten“ werden müssten.

4.2.3 Planung

Unter einem Planungsschritt wird die Bearbeitung der Teilbibliothek Soll verstanden. Dazu wendet der Projektleiter die Planungsvorgaben auf die Teilbibliothek Ist, unter Berücksichtigung der Steuerungsvorgaben aus der Teilbibliothek Strg an. Dadurch werden neue Soll-Elemente erstellt, vorhandene überarbeitet oder entfernt. Dabei ist der Projektleiter pro Planungsschritt nicht gezwungen, alle Vorgaben umzusetzen. Er kann also einen Planungsschritt im Rahmen des Projektcontrolling-Regelkreises auch ganz überspringen und das Soll erst in einem der nächsten Planungsschritte verändern. Abbildung 4.4 zeigt dazu ein Beispiel, bei dem durch Anwenden der beiden Planungsvorgaben Initialplanungen und Bearbeitungszustandsmodell zunächst das Soll-Element se1 und anschließend entsprechende Soll-Versionen Sv11 und Sv12 eingeplant werden.

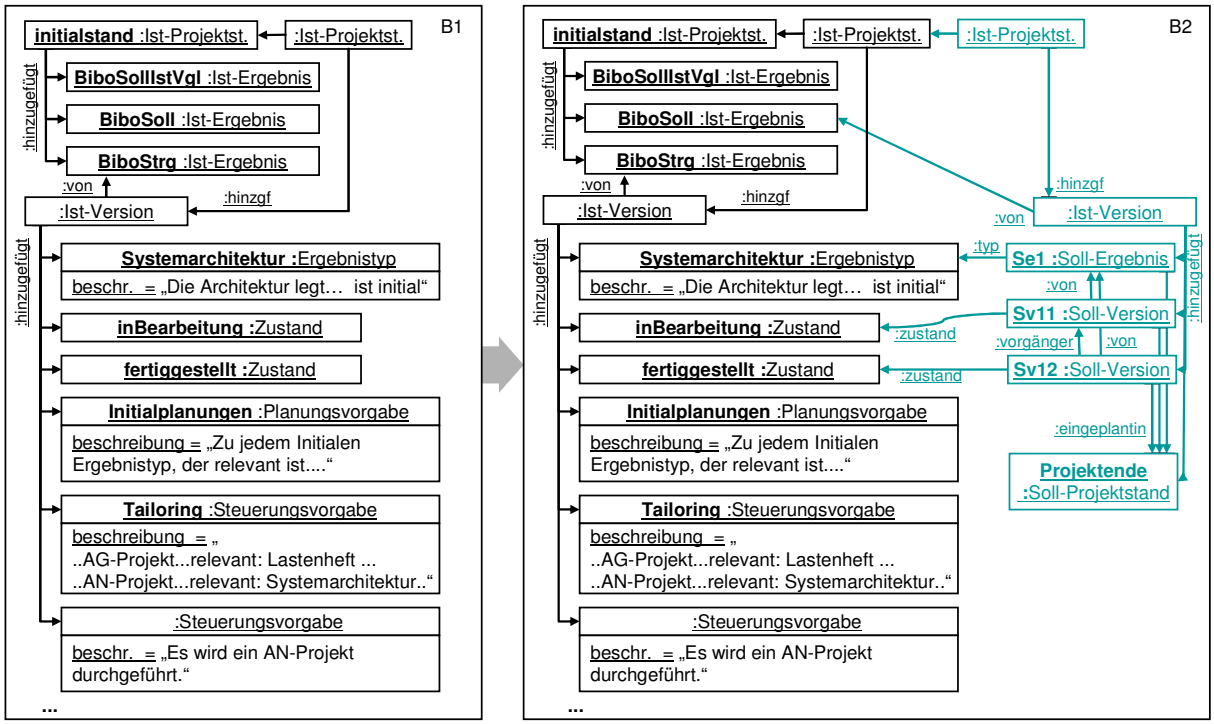


Abbildung 4.4 Beispiel für einen Planungsschritt

Alle Soll-Ergebnisse und -Versionen wurden hier zum einzigen Soll-Projektstand, Projektende, zugeordnet, da die gezeigten Planungsvorgaben diesbezüglich nichts vorgegeben haben. Vorgehensmodelle wie das V-Modell können hier jedoch feingliedrigere Vorgaben haben. Dies wird, um die Darstellung der folgenden Schritte des Projektcontrollings in kompakter Form fortzuführen, in Abbildung 4.5 an einem separaten Beispiel gezeigt.

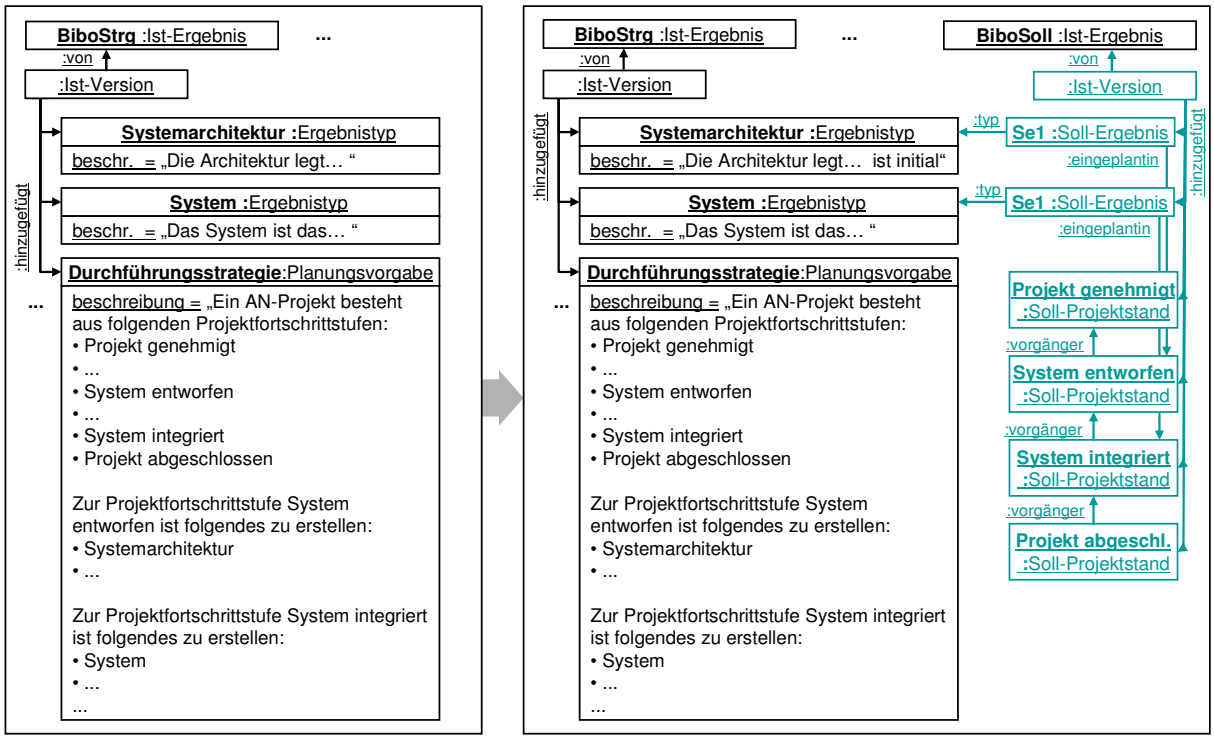


Abbildung 4.5 Beispiel für einen Planungsschritt mit feiner Soll-Projektstands-Planung

Der Planungsschritt lässt sich als eine referenziell nicht transparante Funktion planungsschritt: BIBO $\rightarrow_{Soll}$ BIBO verstehen, wobei die Ausgabe den Planungs- und Steuerungsvorgaben der Eingabe nicht widersprechen darf.
Die Betrachtung der gesamten Produktbibliothek als Eingabe ist notwendig, da einerseits Planungsvorgaben in Abhängigkeit des bisher Erzeugten formuliert werden können sollen, andererseits das dazu auszuwertende Ist-Ergebnis bzw. die gültige Ist-Version (Teilbibliothek Ist) ohne die Betrachtung der Zuordnungen (Teilbibliothek SollIstVgl) nicht eindeutig ermittelt werden kann, was seinerseits auch die Betrachtung der Teilbibliothek Soll nach sich zieht. Ferner können Planungsvorgaben auch auf manuell erzeugten Einträgen des Solls beruhen – etwa der Einplanung einer Projektfortschrittsentscheidung für jeden Soll-Projektstand. Die Steuerungsvorgaben, beispielsweise für das Tailorings, führen auch zur notwendigen Betrachtung der Teilbibliothek Strg.

4.2.4 Durchführung

Unter einem Durchführungsschritt wird die Bearbeitung der Teilbibliothek lst verstanden. Dabei erstellt das Projektteam anhand der Soll-Elemente neue Ist-Elemente. Dabei können in einem Durchführungsschritt beliebig viele, also auch keine, Ist-Elemente erstellt werden. Die Typung der Soll-Elemente verweist auf Vorgehensmodellelemente (ausgenommen Planungs- und Steuerungsvorgaben), die ihrerseits durch Beschreibungen vorgeben, wie dazu passende Ist-Elemente aussehen. Beispielsweise verweist in Abbildung 4.6 das Soll-Ergebnis Se1 auf den Ergebnistyp Systemarchitektur, der wiederum beschreibt, dass darin „HW-, SW- und Externe Einheiten des Systems“ festgelegt werden.

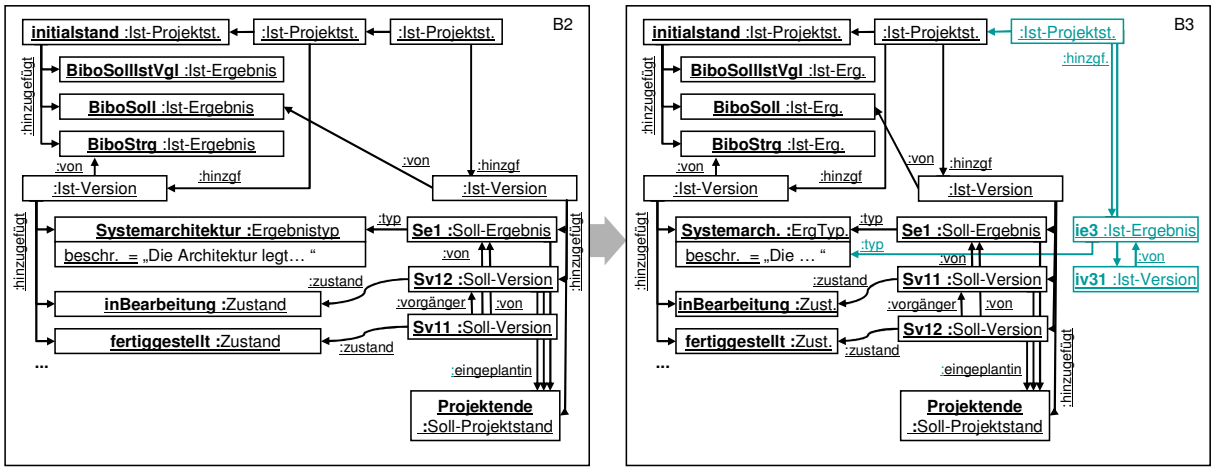


Abbildung 4.6 Beispiel eines Durchführungsschrittes

Der Durchführungsschritt lässt sich als eine referenziell nicht transparante Funktion durchführungsschritt: BIBO $\rightarrow_{lst}$ BIBO[33] verstehen, wobei

- die Ausgabe die Eingabe nur durch das Anlegen neuer Elemente für die Teilbibliothek lst[34] ändern darf, wobei
  - ein neuer Ist-Projektstand zu erzeugen ist, zu dem virtuelle Veränderungen von Ist-Ergebnissen und -Versionen (durch Verwendung der Assoziationen hinzugefügt und gelöscht) direkt zuzuordnen sind
  - die Ergebnisse der Initialproduktbibliothek nicht virtuell gelöscht werden dürfen
- die Ausgabe den Beschreibungen der Ergebnis- und Inhaltstypen aus der Eingabe nicht widersprechen darf.

[33] Ausgenommen sind Planungs- und Steuerungsvorgaben.
[34] Bedingt durch die Betrachtung des speziellsten Typs in der Semantik der Teilbibliothek können damit gerade nicht Elemente der anderen Teilbibliotheken (Strg, Soll und SollIstVgl) verändert werden.

4.2.5 Kontrolle

Unter einem Kontrollschritt wird die Bearbeitung der Teilbibliothek SollIstVgl verstanden. Dazu erstellt der Projektleiter Zuordnungen zwischen Ist- und Soll-Elementen durch Zuordnung derer Versionen. In Abbildung 4.7 wurde beispielsweise die Ist-Version iv31 zur Soll-Version Sv11, und damit das Ist-Ergebnis ie3 zum Soll-Ergebnis Se1 zugeordnet. Zuordnungen werden versioniert, so dass über eine neue Version für das Ist-Ergebnis SollIstVgl bestehende Zuordnungen (virtuell) wieder gelöscht werden können. Dies kann beispielsweise notwendig werden, wenn bestehende Soll- oder Ist-Elemente (virtuell) gelöscht werden und die Bezüge einer Zuordnung dadurch nicht konsistent mit dem aktuellsten Ist-Projektstand sind. Weder in einem Schritt noch im Allgemeinen müssen alle Ist-Versionen zu Soll-Versionen zugeordnet werden. Insbesondere können Ist-Versionen auch in einem abgeschlossenen Projekt unzugeordnet beleiben.

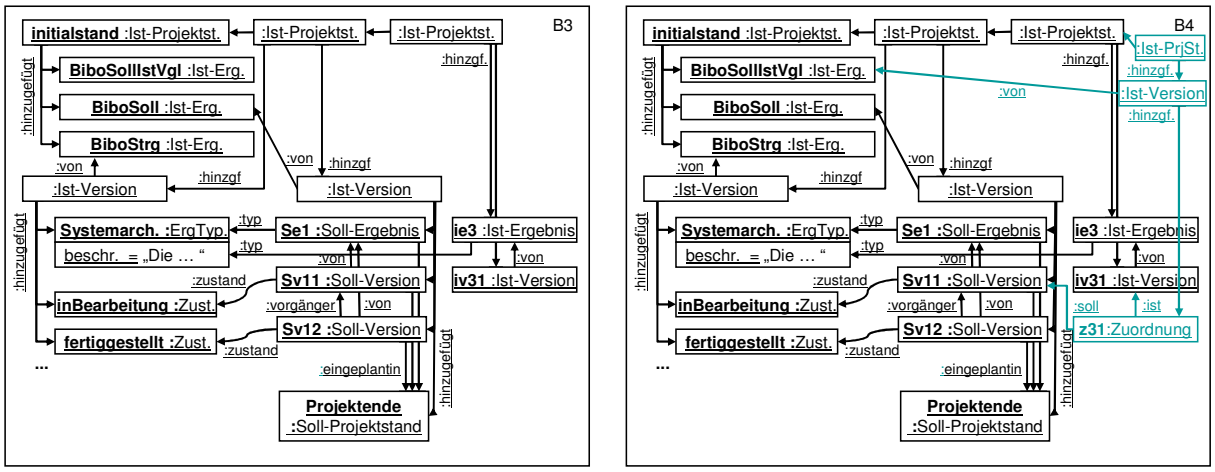


Abbildung 4.7 Beispiel eines Kontrollschritts

Der Kontrollschritt lässt sich als eine referenziell nicht transparante Funktion kontrollschritt: BIBO $\rightarrow_{\text{SollIstVgl}}$ BIBO verstehen, wobei

- zueinander zugeordnete Ist- und Soll-Elemente den gleichen Typ (Ergebnistyp, Zustand) haben müssen,
- zueinander zugeordnete Ist- und Soll-Elemente im aktuellsten Projektstand vorhanden sein müssen
- alle Ist-Versionen eines Ist-Ergebnisses zu Soll-Versionen desselben Soll-Ergebnisses zugeordnet werden müssen,

4.3 Aufgabenstellung: Operationalisierung

Basierend auf der im Kapitel 4.2 definierten Problemdomäne sind folgende Teilprobleme in dieser Arbeit im Allgemeinen zu lösen:

I. Die Anwendung von Steuerungsvorgaben, die keine Benutzerentscheidungen erfordern, ist zu automatisieren. Formal entspricht dies der Definition einer Funktion (steuerung in Abbildung 4.8), die innerhalb eines Steuerungsschrittes einmalig oder auch mehrmals[35] angewendet werden kann.

II. Die Anwendung von Planungsvorgaben, die keine Benutzerentscheidungen erfordern, sind zu automatisieren. Formal entspricht dies der Definition einer Funktion (planung in Abbildung 4.8), die innerhalb eines Planungsschrittes einmalig oder auch mehrmals angewendet werden kann.

III. Die Zuordnung von Ist-Versionen zu Soll-Versionen ist durch eine automatisierte Ermittlung von Zuordnungsvorschlägen zu unterstützen. Weiterhin ist die Bestimmung und Entfernung ungültig gewordener Zuordnungen zu automatisieren. Formal entspricht dies der Definition der beiden Funktionen $kontrolle_V$ und kontrolle (in Abbildung 4.8), die innerhalb eines Kontrollschritts einmalig oder auch mehrmals angewendet werden können[36]. Die erstere erzeugt dabei Vorschläge zur Änderung der Teilbibliothek für die Kontrolle in Form eines „Deltas“. Aus diesem Delta wählt der Projektleiter diejenigen Vorschläge aus, die er umsetzen möchte, wobei die Umsetzung jener Vorschläge der zweiten Funktion entspricht.

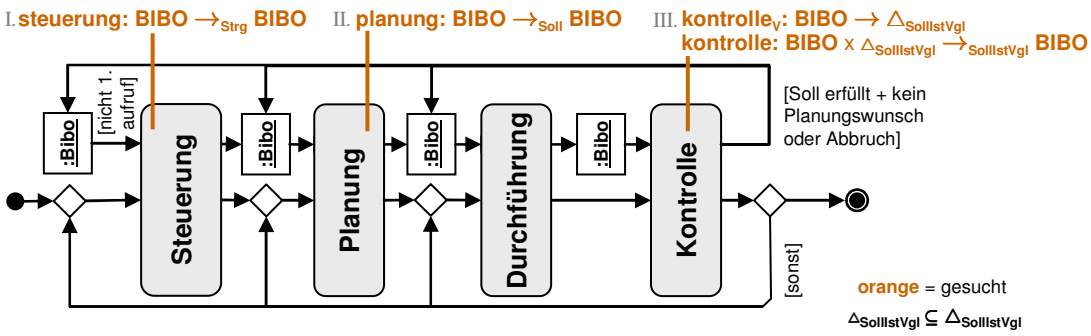


Abbildung 4.8 Aufgabenstellung

[35] Da der entsprechende Schritt des Projektcontrollings auch mit der gesuchten Funktion nicht vollständig formalisiert wird, stellt sich die Frage, wie die Anwendung jener Funktion mit der restlichen manuellen Arbeit in jenem Schritt interagiert. In dieser Problemstellung sollen hierzu keine Einschränkungen getroffen werden, so dass die Funktion zur Automatisierung auch mehrfach innerhalb des Schrittes ausgeführt werden können soll. Jede Ausführung muss dabei so beschaffen sein, dass zwischenzeitliche, manuelle Änderungen nicht verloren gehen.

[36] Der Index V in $kontrolle_V$ steht für *V*orschläge

5 Lösung

In diesem Kapitel wir die Lösung der Problemstellung vorgestellt. In Kapitel 5.1 wird der Lösungsansatz vorgestellt. Die detaillierte Lösung wird im Rest dieses Kapitels 5 allerdings nicht genau in die drei Fragen bezüglich der Steuerung, Planung und Kontrolle gegliedert, da es umfangreiche Querschnitte gibt. Dementsprechend wird folgende Einteilung (vgl. Abbildung 5.1) gewählt:

- Die **Grundlagenerweiterungen** (Kapitel 5.2) umfassen Konzepte zur Erweiterung der Inkrementellen Transformation. Diese sind unabhängig von der Problemdomäne des Projektcontrollings und werden in entsprechend allgemeiner Form dargestellt. Dazu gehört das Verwenden von um Constraints erweiterte Produktionsregeln, die Möglichkeit, Graphelemente mit bestimmten Ids zu erzeugen und Suchmuster auf diese einzuschränken, sowie ein neues Szenario – die induktiv inkrementelle Transformation: Diese erlaubt es, per Transformation erzeugte Graphelemente wiederum als Eingabe für die folgenden Transformationsanwendungen zu verwenden.
- Das **Automatisierungskonzept** (Kapitel 5.3) umfasst die formalen Teile der Operationalisierung. Dazu gehören Erweiterungen des Produktbibliotheksmodells sowie die formal ausführbaren Funktionen steuerung, planung, kontrolle. Kernidee dabei ist, Vorgehensmodellvorgaben in Form virtualisierter Produktionsregeln als Teil der Produktbibliothek zu führen. So kann sowohl die Produktbibliothek als auch die Vorgaben selbst Gegenstand der inkrementellen Transformation sein. Diese kann auch für die Kontrolle eingesetzt werden, indem das Auswählen von Vorschlägen als eine vom Projektleiter manuell ausgeführte Redexauswahlfunktion angesehen wird.
- Die **Integrationsmethodik** für Prozessbeschreibungssprachen (Kapitel 5.4) umfasst den methodischen Teil zum Einsatz des Automatisierungskonzepts in der Praxis. Es wird gezeigt, wie für einen Prozessingenieur geeignete Sprachkonstrukte entwickelt, und mit einer auf Produktionsregeln basierenden Semantik unterlegt werden kann, die wiederum dem Automatisierungskonzept zuführbar ist. Zur Sprachbeschreibung werden UML-Aktivitätsdiagramme, für die Semantikabbildung Transformationen eingesetzt, die Modelle und Produktionsregeln erzeugen können.

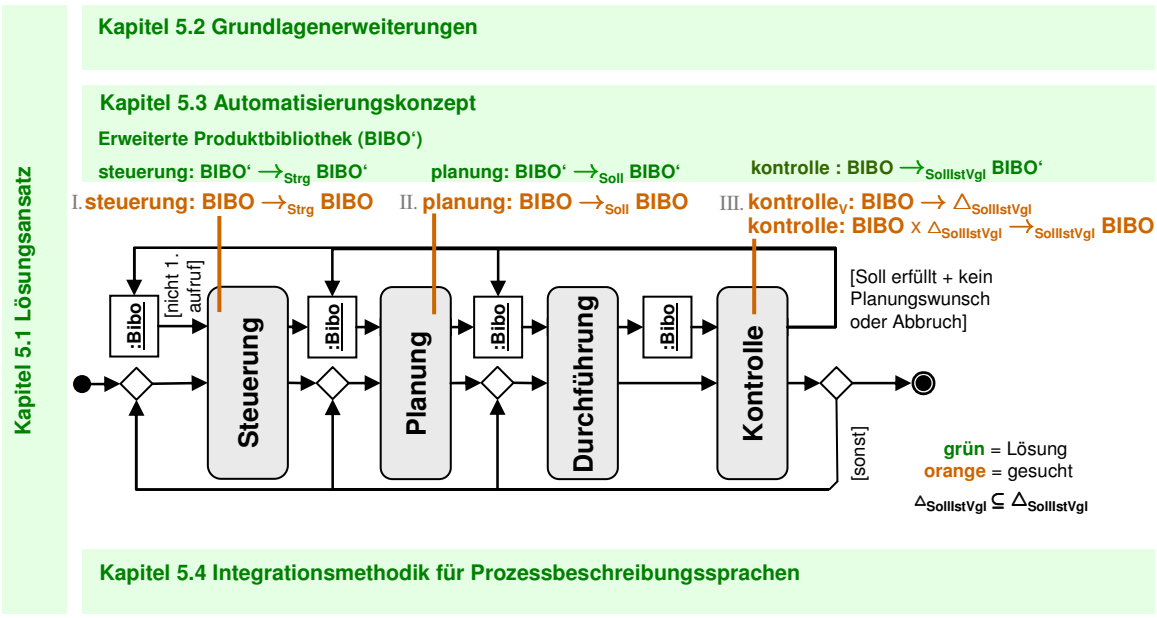


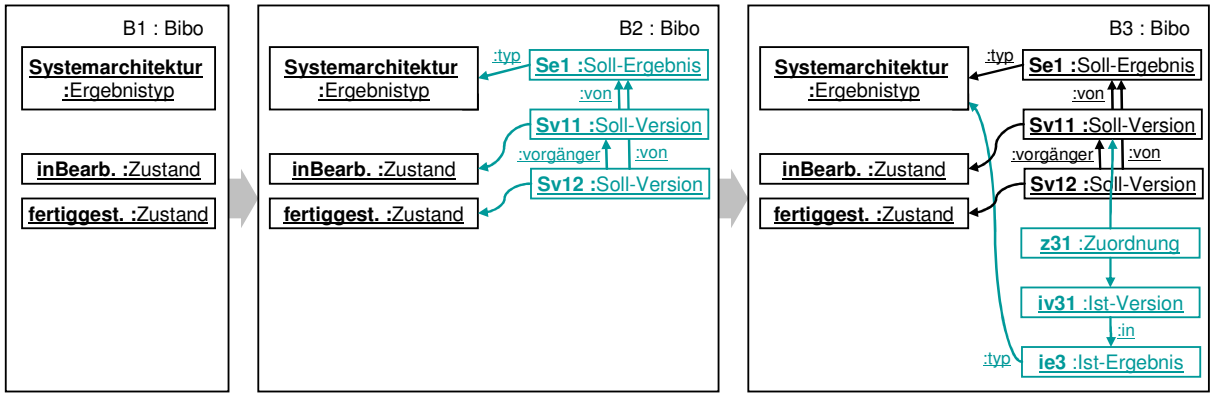
Abbildung 5.1 Lösungsbestandteile und Überblick zum Aufbau des Lösungskapitels

5.1 Lösungsansatz

In diesem Kapitel wird die Lösung im Überblick vorgestellt. Die Unterkapitel 5.1.1 bis 5.1.3 skizzieren das Automatisierungskonzept, Kapitel 5.1.4 die Integrationsmethodik. Auf die jeweils notwendigen Grundlagenerweiterungen wird an der jeweils relevanten Stelle eingegangen.

5.1.1 Automatisierung der Planung

Naive Implementierung. Ein naheliegender Lösungsansatz für die Bereitstellung der in der Aufgabenstellung gesuchten Funktion planung wäre, diese in Form eines SW-Programms zu implementieren. Eine solche Implementierung darf jedoch nicht beliebig sein: Würde beispielsweise das Soll aus dem aktuellen Ist bei jeder Ausführung des SW-Programms von Grund auf neu erstellt, würden die zwischenzeitlich (in einem Kontrollschritt) manuell erstellten Zuordnungen verloren gehen. Abbildung 5.2 erläutert dies konkret an einem Beispiel.



B1: Bei Projektstart liegt in der Produktbibliothek nach dem ersten Steuerungsschritt nur das Vorgehensmodell vor.

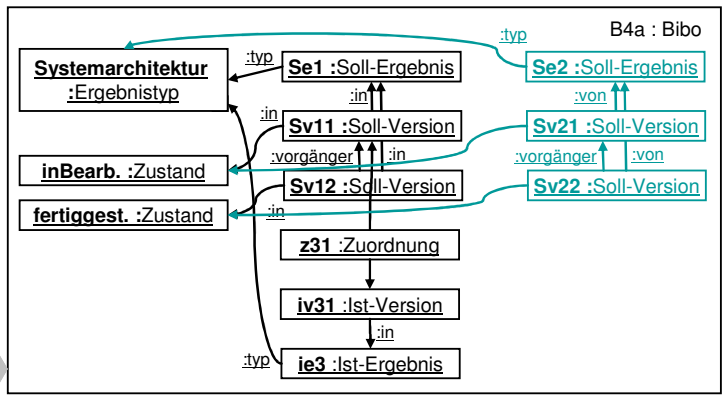
B2: Anhand des Vorgehensmodells sei in dem darauffolgenden Planungsschritt das Soll durch eine automatische Berechnung (z.B. Transformation) generiert.

B3: Im anschließenden Durchführungsschritt werden einige Ist-Elemente per Hand erarbeitet und im darauf folgenden Kontrollschritt per Hand den Soll-Elementen zugeordnet.

Nun soll ein erneuter Planungsschritt folgen...

B4a: Wird (wieder) eine einfache Generierung des Solls durchgeführt, so entstehen u.a. Duplikate im Soll. Es müsste per Hand entschieden werden, welche neu erzeugten Soll-Elemente Duplikate sind, um diese entsprechend zu entfernen.

B4b: Wird, als Alternative zu B4a, vor der Generierung des Solls das alte Soll entfernt, gehen die Zuordnungen verloren. Für z31 ist der Bezugsversion Sv11 nicht mehr vorhanden; es müsste per Hand auf eines der neuen Soll-Versionen umgehoben werden.



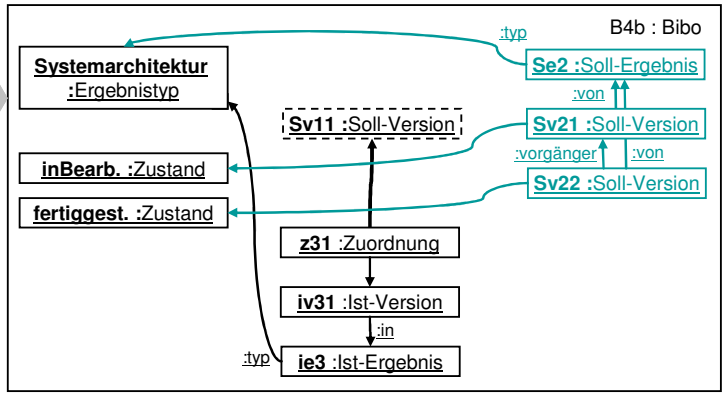


Abbildung 5.2 Herausforderung bei der Automatisierung der Planung

Inkrementelle Transformation. Mit dem Werkzeug der inkrementellen Transformation kann prinzipiell erreicht werden, dass das Soll nicht bei jeder Berechnung von Grund auf neu erstellt wird, sondern effektiv nur die daran notwendige Änderungen vorgenommen werden. Doch wie kann die inkrementelle Transformation konkret für die Problemdomäne des Projektcontrollings angewendet werden? Die inkrementelle Transformation basiert auf dem Transformieren von Graphen mittels Produktionsregeln. In der Problemstellung ist die Struktur einer Produktbibliothek (Abbildung 4.1) als ein Graph eingeführt. Planungsvorgaben sind als Texte in natürlicher Sprache verfasst, die nicht automatisiert als eine Transformation (zur Berechnung eines neuen Solls) angewendet werden können.

Planungsvorgaben als Produktionsregeln. Unter Berücksichtigung der in Abbildung 5.2 gezeigten Herausforderung, und in Anbetracht der diesbezüglichen Mängel existierender Prozessbeschreibungssprachen (siehe Kapitel 3.2) werden Planungsvorgaben als Produktionsregeln formalisiert. Allerdings sind Planungsvorgaben über den Verlauf eines Projektes nicht fest, sondern können zwecks Ausgestaltung selbst Gegenstand von Steuerungsvorgaben sein. Somit müssen Vorgaben einerseits Teil der Produktbibliothek sein, und andererseits in Form eines Graphen vorliegen, um von Steuerungsvorgaben analysiert werden zu können (vgl. Kapitel 2.7.2). Abbildung 5.3 zeigt die Lösung, die darin besteht, die als Produktionsregeln formalisierten Planungsvorgaben in virtualisierter Form als Teil der Produktbibliothek abzulegen. Um etwas Flexibilität zu ermöglichen (konkret um eine Planungsvorgabe auch mit mehreren Produktionsregeln formalisieren zu können), werden Produktionsregeln den Planungsvorgaben zugeordnet. Bei der inkrementellen Transformation einer Produktbibliothek `B` werden alle darin enthaltenen Produktionsregeln, die einer Planungsvorgabe zugeordnet sind, aus `B` devirtualisiert und auf `B` angewendet. Zu beachten ist, dass mehrfache Devirtualisierungen einer Produktionsregel zum identischen Ergebnis führt, so dass die Einträge in der Transformationshistorie ihre Bezüge nicht verlieren.

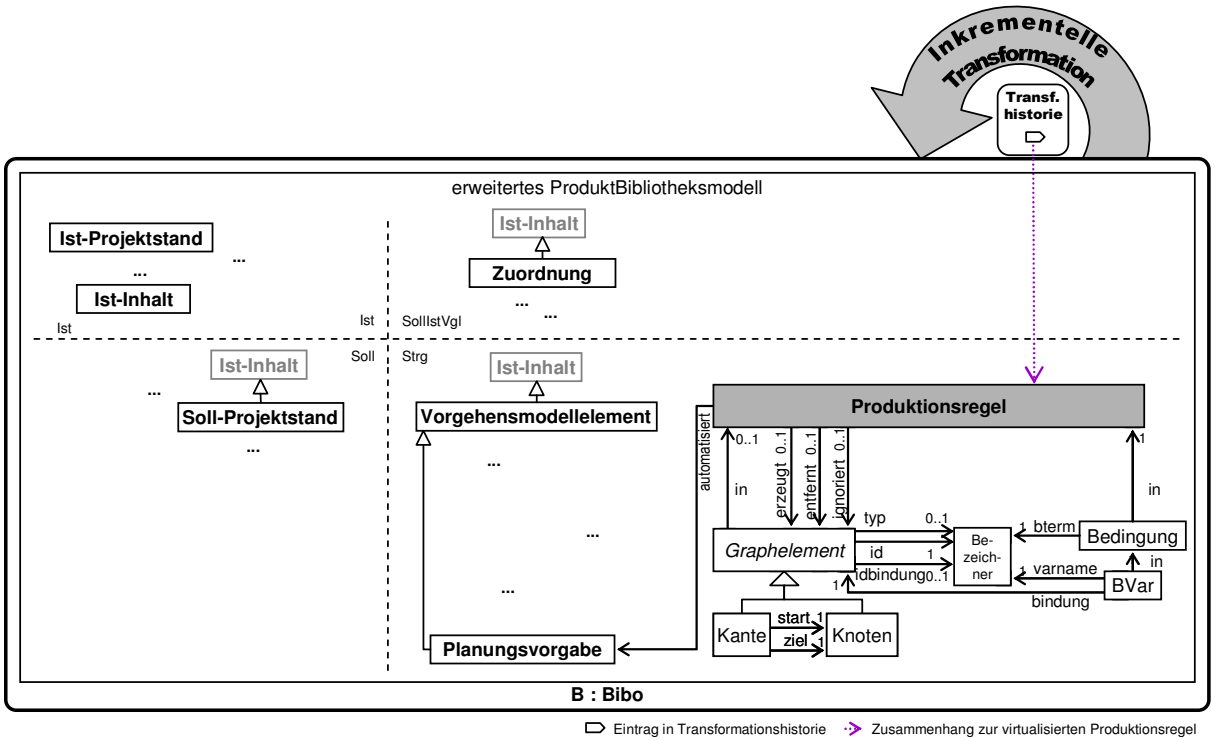


Abbildung 5.3 Vorgaben als virtualisierte Produktionsregeln in einer Produktbibliothek

Bezug auf konkrete Vorgehensmodellelemente. In nahezu allen Planungsvorgaben werden bestimmte Elemente des Vorgehensmodells konkret adressiert. In der folgenden

Planungsvorgabe sind das beispielsweise die zwei Ergebnistypen Prüfspezifikation und Prüfprozedur sowie der Inhaltstyp Prüffall:

> „Für jeden spezifizierten Prüffall wird eine Prüfprozedur […] erstellt."

Mit einer „normalen" Produktionsregel lassen sich jedoch konkrete Ergebnistypen nicht ohne Weiteres referenzieren: Denn bei der Belegung des Suchmusters einer Produktionsregel werden nur die Typen der Musterelemente berücksichtigt – nicht etwa deren Id. Anders formuliert werden sämtliche Musterelemente als Variablen behandelt und nicht als Referenzen auf konkrete Knoten in dem zu transformierenden Graphen. Damit würde beim Ausführen des Pattermatchings z.B. der Knoten Prüfspezifikation (Abbildung 5.4) mit *allen* Ergebnistypen des Projektes belegt werden können – nicht nur mit denen vom beabsichtigten Ergebnistyp. Um dies dennoch zu ermöglichen, wird das Konzept des Id-Patternmatchings (Kapitel 5.2.1) definiert und in die inkrementelle Transformation integriert.

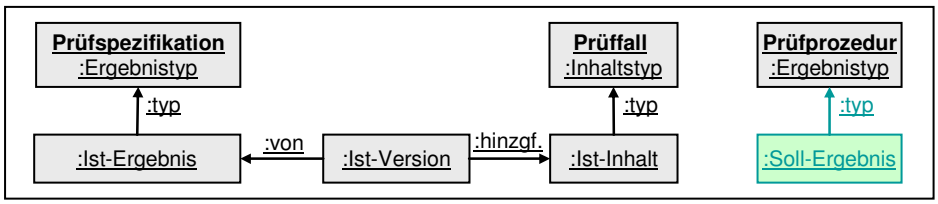


Abbildung 5.4 Bezüge zu konkreten Vorgehensmodellelementen

Bezug auf konkrete Versionen und Inhalte. Der Bezug auf die aktuellste Version eines Ergebnisses (und dessen Inhalte) wurde in der Problemstellung algebraisch spezifiziert. Diese in Produktionsregeln umzusetzen ist insofern schwierig, als dass, falls dazu nicht genau eine einzige Produktionsregel verwendet wird, die Gefahr von unbeabsichtigten Wechselwirkungen mit den Produktionsregeln anderer Planungsvorgaben besteht (vgl. Isolationseigenschaft von Transaktionen [89,90]).

Durch die Verwendung von OCL-Constraints lässt sich die Transaktionsthematik für den Zugriff auf Versionen und Inhalte umgehen: Wie Abbildung 5.5 zeigt, können komplexere Bedingungen als Teil einer Produktionsregel, und damit atomar, untergebracht werden.

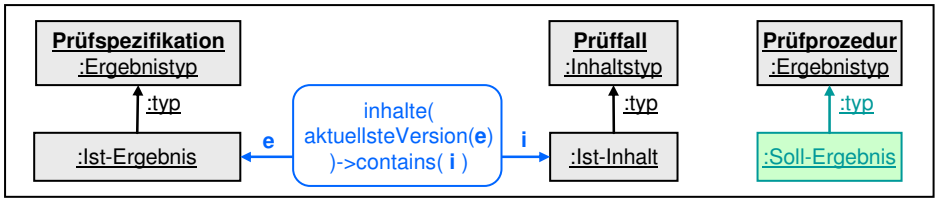


Abbildung 5.5 Zugriff auf Versionen und Inhalte

Neben der Transaktionsthematik vereinfachen OCL-Funktionen auch die Definition korrekter Produktionsregeln. So greift beispielsweise die Produktionsregel in Abbildung 5.4 auch dann, wenn ein Prüffall in einer späteren Version (virtuell) gelöscht wurde. Durch Nutzung von Constraints mit entsprechenden OCL-Funktionen kann der korrekte Zusammenhang zwischen Inhalt und Ergebnis auf einfache Weise, d.h. insbesondere ohne zusätzliche Produktionsregeln, formuliert werden (Abbildung 5.5). Auch für den umgekehrten Fall – bei dem eine Produktionsregel auch dann noch greifen soll, wenn neue Versionen erstellt werden – lässt sich dieser Ansatz anwenden. Abbildung 5.6 zeigt, wie dies im Rahmen von Meilensteinen und Iterationen aussieht.

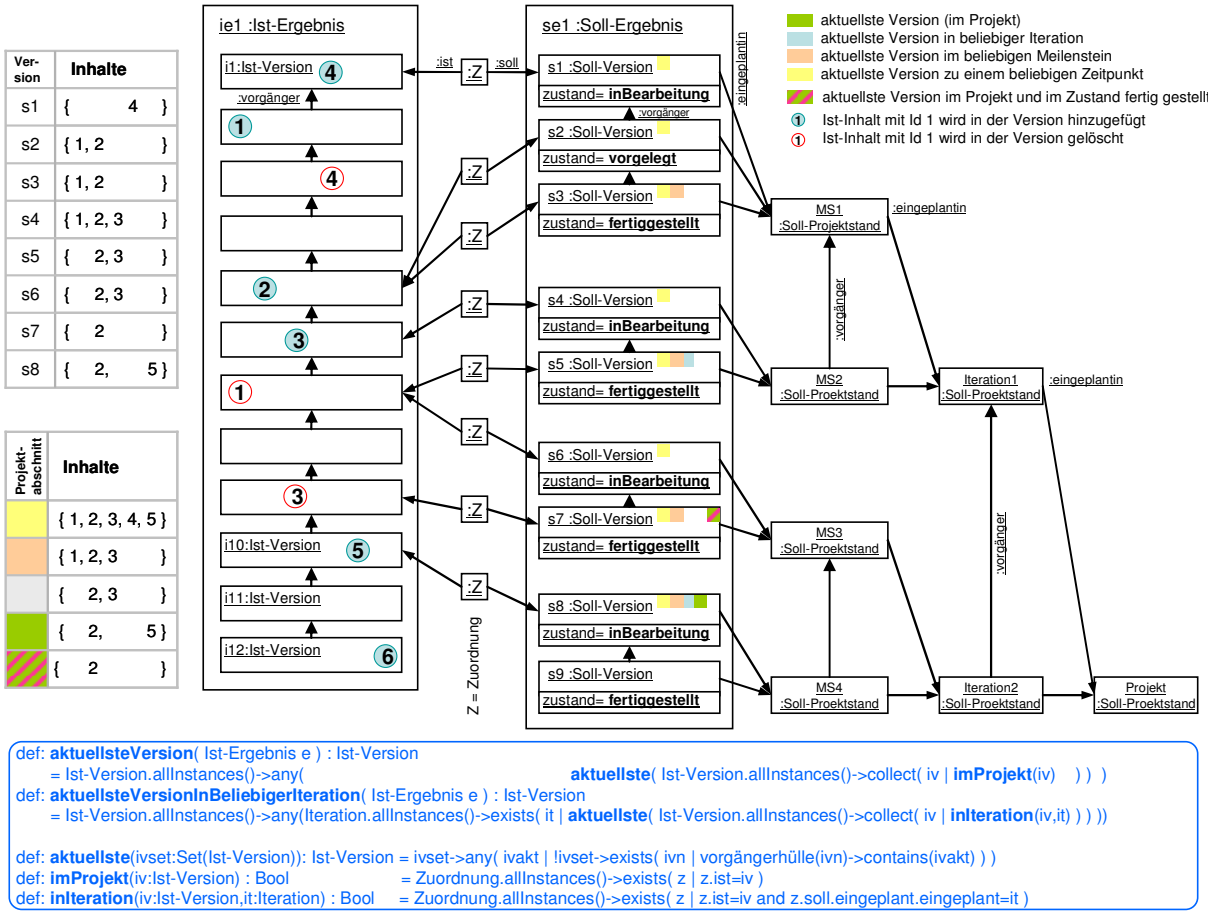


Abbildung 5.6 Bezüge zu Projektabschnitten

Im Hauptteil der Abbildung ist eine Projektsituation dargestellt, in der zum Soll-Ergebnis se1 – bestehend aus den Soll-Versionen s1 bis s9 – das Ist-Ergebnis ie1 – bestehend aus den Ist-Versionen i1 bis i12 – zugeordnet wird. Konkret werden von ie1 nur die ersten zehn Ist-Versionen zugeordnet, so dass die Soll-Version s9 ohne Zuordnung bleibt. Weiterhin liegt über den Soll-Versionen eine Soll-Projektstand-Struktur aus Meilensteinen und Iterationen vor, die in höchster Instanz das gesamte Projekt abdecken. Nach dieser Struktur können die Soll-Versionen bezüglich ihrer Aktualität in verschiedene Projektabschnitte klassifiziert werden: Grün markiert ist diejenige Soll-Version, die im gesamten Projekt die aktuellste ist (vgl. Abbildung 5.5), zu der im unteren Teil der Abbildung 5.6 die dazu passende OCL-Funktion (aktuellsteVersion) angegeben ist. Blau markiert ist jede Soll-Version, die in einer beliebigen Iteration die aktuellste ist. Wiederum ist im unteren Teil der Abbildung 5.6 die dazu passende OCL-Funktion (aktuellsteVersionInBeliebigerIteration) angegeben. Wird diese Funktion in Abbildung 5.5 verwendet (anstelle von aktuellsteVersion), dann bleiben auch solche Prüfprozeduren erhalten, die es z.B. nur in der ersten von vielen Iterationen gab. Nach gleicher Methode können auch die anderen Projektabschnitte definiert werden.

Zustandsübergänge als Produktionsregeln. Während mit automatisierten Planungsvorgaben Soll-Ergebnisse erzeugt werden, obliegt es dem automatisierten Zustandsmodell, neue Soll-Versionen zu erzeugen. In Analogie zur Automatisierung der Planungsvorgaben liegt es daher nahe, pro Zustandsübergang genau eine Produktionsregel vorzusehen, die bei Erfüllung einer Zustandsübergangsbedingung die jeweils als nächstes anstehende Soll-Version erzeugt.

Da das Erzeugen von Soll-Versionen für alle Zustandsübergänge gleichermaßen gilt, braucht mit der zu einem Zustandsübergang zugeordneten Produktionsregel nur die

Zustandsübergangsbedingung formalisiert werden (die im Sinne eines Prädikats ein „Ja“ oder „Nein“ erzeugt). Die effektive, für die inkrementelle Transformation verwendete Produktionsregel kann aus der Zustandsübergangsbedingung und einer Schablone (Abbildung 5.7) durch eine Graphaddition (nach Kapitel 2.2.1) konstruiert werden: Die erste Schablone aus Abbildung 5.7 wird für die Einplanung der ersten Soll-Version, die zweite für alle folgenden Soll-Versionen pro Soll-Ergebnis und -Projektstand eingesetzt. Mit den durch die Graphaddition entstehenden Erweiterungen kann auch eingeschränkt werden, bezüglich welcher Soll-Projektstände Soll-Versionen erzeugt werden. Beispielsweise kann die Belegung des Suchmusterknotens für Soll-Projektstände auf solche der untersten Hierarchiestufe (den Meilensteinen) beschränkt werden, um die Situation aus Abbildung 5.6 nachzubilden. Zusätzlich werden die mit einem Fragezeichen markierten Suchknoten an die konkreten Zustände des betroffenen Zustandsübergangs und an den vom Zustandsmodell betroffenen Ergebnistyp gebunden.

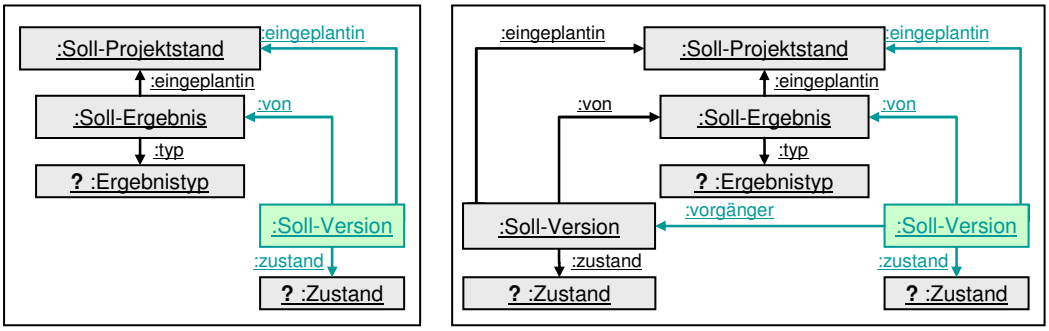


Abbildung 5.7 Skizze der Produktionsregel-Schablonen für Soll-Version-Planung

Je nach Zustandsübergangsbedingung würden mit diesem Ansatz Soll-Versionen jedoch erst dann eingeplant, wenn das zugehörige Ist-Element die jeweils bisherige Soll-Version erfüllt hat. Damit würde die automatisierte Planung nur jeweils einen Schritt im Voraus ablaufen, obwohl bestimmte Soll-Versionen, bedingt durch das Zustandsmodell, auch viel früher eingeplant werden könnten – z.B. noch bevor das entsprechende Ist-Element überhaupt erzeugt wurde. Ein konkretes Beispiel ist die Einplanung der Vorlage und der anschließenden Fertigstellung eines Ergebnisses. Dies könnte auch über mehrere Meilensteine oder Iterationen in die Zukunft reichen, was wiederum erlaubt, die Planung bereits weit im Vorfeld genauer zu gestalten, und somit eine bessere Grundlage für Aufwandschätzungen etc. zu liefern. Bei der manuellen Planung ist dies ohne Weiteres möglich – mit der Automatisierung durch eine Zustandsmaschine jedoch nicht.

Optimistische Soll-Version-Planung. Um Soll-Versionen im Voraus automatisiert einplanen zu können, wird das Konzept der *optimistischen Soll-Version-Planung* eingeführt. Sie besteht in einer besonderen Behandlung der Situation, in der sich die Zustandsmaschine zur Einplanung der Soll-Versionen in einem Zustand befindet, zu dem keines der abgehenden Zustandsübergänge aktuell schaltet. Anstatt, wie bei einer Zustandsmaschine, zu warten bis sich dies ändert, wird eines der abgehenden Zustandsübergänge optimistisch durchgeführt: Die nicht erfüllte Zustandsübergangsbedingung eines gewählten Zustandsübergangs wird unter der Voraussetzung ignoriert, dass keine Zustandsübergangsbedingung der alternativen Zustandsübergänge erfüllt ist. Durch Nutzung der inkrementellen Transformation ist sichergestellt, dass optimistisch eingeplante Soll-Versionen automatisch abgebaut werden, sobald sich der Optimismus als unberechtigt erweist. Für die Auswahl des in der gegebenen Situation optimistisch durchzuführenden Zustandsübergangs wird das Produktbibliotheks-modell um eine entsprechende Markierungsmöglichkeit erweitert.

5.1.2 Teilautomatisierung der Kontrolle

Naive Implementierung. Anders als bei der Automatisierung der Vorgaben, kann die Berechnung der Zuordnungsvorschläge prinzipiell in Form eines SW-Programms implementiert werden, da die erzeugten Zuordnungsvorschläge per Hand in persistente Zuordnungen überführt werden. Eine mehrmalige Ausführung eines entsprechenden SW-Programms verursacht also nicht einen Verlust von Informationen. Allerdings entsteht vielmehr ein gegenteiliger Effekt, ein Überangebot an Informationen, wenn eine Implementierung beliebig gewählt werden darf. So sind typischerweise pro Kontrollschritt nur solche zum Produktbibliotheksmodell passende Zuordnungsvorschläge interessant, die

(i) sich auf Soll-Elemente beziehen, zu denen noch kein Ist-Element zugeordnet ist,
(ii) nur die nächste offene Soll-Version mit einer Ist-Version abdecken, und
(iii) die nicht in einem Kontrollschritt als unzutreffend eingeschätzt wurden.

Die erste Einschränkung zielt auf das „Delta“ der Projektdurchführung ab: der Projektleiter möchte typischerweise nur die jeweils neuen Ergebnisse (und deren Version) zum Soll zuordnen. Dies schließt jedoch nicht aus, bisherige Zuordnungen zu entfernen. Die dadurch nicht mehr zugeordnete Ergebnisse (und deren Versionen) sind dann – für die Erzeugung von Zuordnungsvorschlägen – einfach als „neu“ zu betrachten.

Die zweite Einschränkung zielt auf die Situation ab, in der ein Ist-Element auf mehrere Soll-Elemente passt. In Abbildung 5.8 passt das neue Ist-Ergebnis Ie3 auf die Soll-Ergebnisse Se1 und Se2. Die Wahl der Zuordnung für die erste zu belegende Soll-Version (entweder Sv11 oder Sv21) schließt Zuordnungsvorschläge für die nachfolgenden Versionen aus.

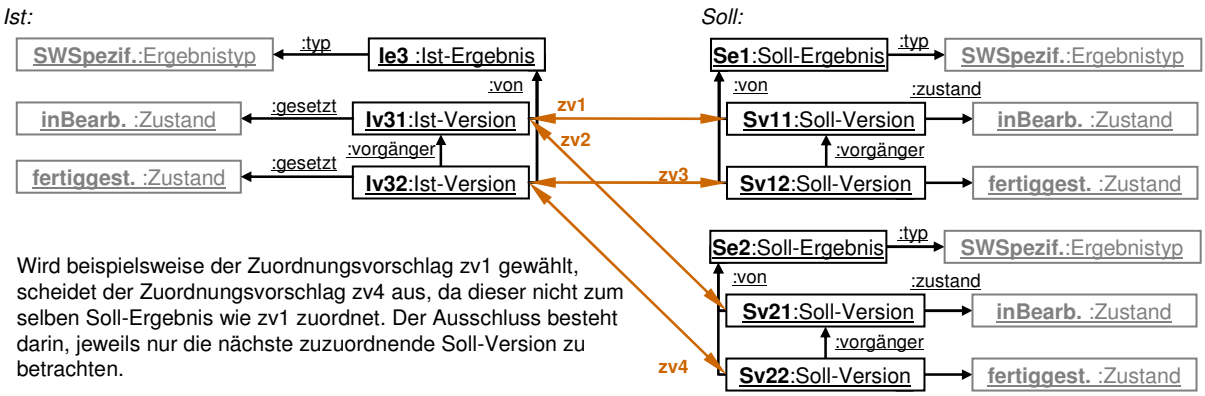


Abbildung 5.8 Ausschluss angebotener Zuordnungsvorschläge auf Ergebnisebene

In Abbildung 5.9 wird die Ebene der Versionen innerhalb eines Ergebnisses betrachtet, wobei die Ist-Version Iv31 auf die Soll-Versionen Sv12 und Sv16 passt. Durch die Einschränkung der Zuordnungsvorschläge auf die nächste zu belegende Version (hier Sv12) kann auch in dieser Situation die Zahl der von einem Projektleiter zu betrachtenden Vorschläge pro Zuordnungsentscheidung reduziert werden. Um Vorschläge für die nachfolgenden Soll-Versionen zu erhalten ist die Berechnung entsprechend mehrmals auszuführen – beispielsweise nach jeder einzelnen getroffenen Zuordnung. Dieses Vorgehen ist besonders dann wirkungsvoll, wenn es viele ähnliche Soll-Ergebnisse und/oder viele Soll-Versionen gibt. Beides ist für Softwareprojekte typischerweise gegeben: Komplexe Softwaresysteme bestehen aus vielen Bestandteilen, was zur hohen Zahl ähnlicher Soll-Ergebnisse führt. Iterationen im Kleinen (Qualitätssicherung) und im Großen (Inkrementelle bzw. Evolutionäre Entwicklung [91]) führen zu einer hohen Zahl ähnlicher Soll-Versionen.

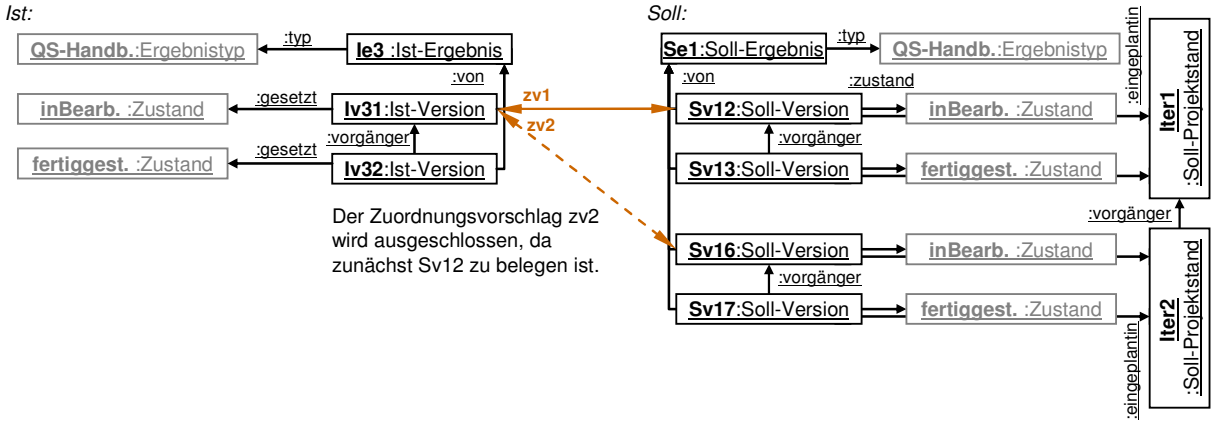


Abbildung 5.9 Ausschluss angebotener Zuordnungsvorschläge auf Versionsebene

Die dritte Einschränkung zielt darauf ab, zu als fehlerhaft bzw. nicht zu berücksichtigen bewerteten Ist-Versionen keine Zuordnungsvorschläge zu erzeugen. Dieser Fall tritt ein, wenn Ist-Versionen nicht genau dem eingeplanten Verlauf des Zustandsmodells, und damit den Soll-Versionen, folgen. Abbildung 5.10 zeigt dazu eine typische Projektsituation als Beispiel. Konkret ist das Ist-Ergebnis Ie3 in seiner ersten Version im Zustand in Bearbeitung gewesen, dann einmal im Zustand fertig gestellt, zweimal im Zustand vorgelegt und ist schließlich in den aktuellsten zwei Versionen wieder im Zustand fertig gestellt. Anhand dieser Informationen ist jedoch nicht entscheidbar, ob für die Soll-Version Sv12 die Zuordnung zv1, zv2 oder sogar keine von beiden zu wählen ist.

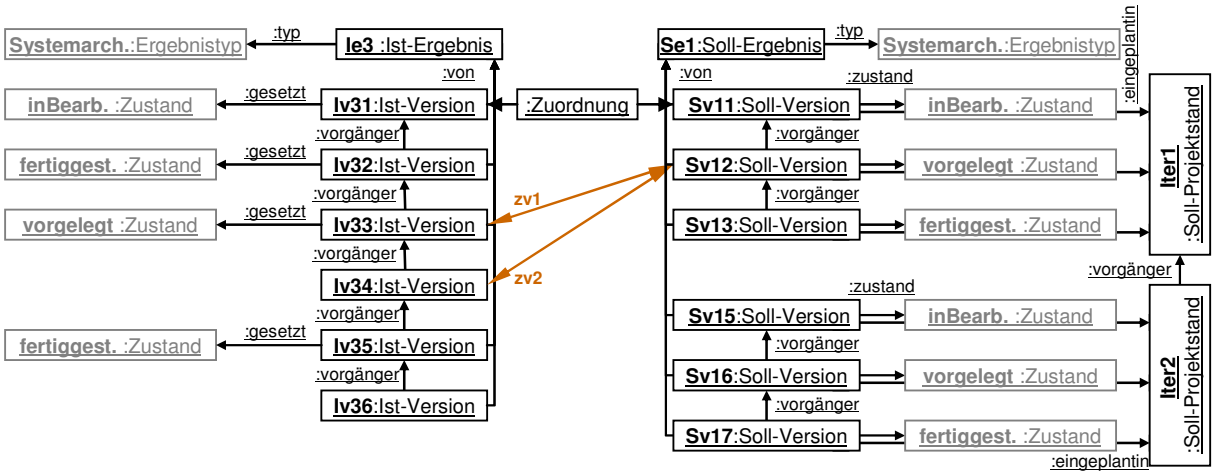


Abbildung 5.10 Ausschluss angebotener Zuordnungsvorschläge durch Fehlerauswertung

Um eine korrekte Entscheidung zu treffen, muss der Projektleiter Informationen über den Hergang besitzen. Die Konstellation in Abbildung 5.10 könnte sich beispielsweise wie folgt zugetragen haben:

- Iv31: Der Systemarchitekt erstellt als Mitglied des Projektteams in einem Durchführungsschritt die erste Version der Systemarchitektur, und setzt diese korrekt auf den Zustand in Bearbeitung.
- Iv32: Der Systemarchitekt erstellt eine weitere Version der Systemarchitektur, und setzt diese in den Zustand fertig gestellt.
- Iv33: Der Projektleiter bemerkt, dass – entgegen der Planung – keine eigenständige Qualitätssicherung durchgeführt wurde, und veranlasst den Systemarchitekten die Systemarchitektur in einer neuen Version und entsprechendem Zustand vorzulegen.

- lv34: Dem Systemarchitekten fällt kurz[37] nach dem Vorlegen ein, dass er doch noch etwas vergessen hatte. Dies führt er mit der Version lv34 nach, die nun die „tatsächlich“ zu prüfende Version ist.
- lv35: Nach erfolgreicher Prüfung kann die Systemarchitektur in dieser Version lv35 als fertig gestellt gelten.
- lv36: Der Systemarchitekt beginnt mit den Arbeiten für die zweite Iteration des Projektes (Iter2), vergisst aber, den Zustand auf „in Bearbeitung“ zurückzusetzen.

Anhand dieses Hergangs kann für die Soll-Version Sv12 der Zuordnungsvorschlag zv2 als der zu wählende bestimmt werden. Wie der Hergang bezüglich der Ist-Versionen lv35 und lv36 zeigt, gilt jedoch nicht, dass es immer die aktuellere Ist-Version (bei zwei aufeinander folgenden im gleichen Zustand) ist. Im Allgemeinen kann selbst eine Kette von Ist-Versionen, die genau dem Zustandsmodell folgt, durch den Hergang für die Zuordnungsvorschläge zu ignorieren sein. Zusammengefasst ist also ein Mechanismus nötig, der jede Ist-Version von den Zuordnungsvorschlägen ausschließen kann, und diese Entscheidungen persistiert, um sie nicht bei jedem Kontrollschritt erneut durch den Projektleiter erfassen zu lassen.

Inkrementelle Implementierung. Als Grundlage für die Umsetzung der im vorangegangen Paragraph erörterten Beschränkungen wird das Produktbibliothekmodell erweitert: Im Teilmodell SollIstVgl wird ein Prädikat (das neue Konstrukt Ausschluss) eingeführt, welches Ist-Versionen referenzieren kann. Jede so referenzierte Ist-Version wird dann bei der Berechnung der Zuordnungsvorschläge nicht berücksichtigt. Alle anderen Einschränkungen können dagegen mit den bisherigen, von einer Produktbibliothek bereitgestellten Informationen umgesetzt werden. Die Umsetzung selbst kann damit in Form eines SW-Programms erfolgen.

Die sich insgesamt ergebende Implementierung würde sich vom Konzept her der inkrementellen Transformation stark ähneln: Die ausgelesenen und modifizierten Zuordnungen, Soll- und Ist-Elemente (einschließlich des neuen Prädikats) entsprächen der „Transformationshistorie“. Die Erzeugung neuer Zuordnungsvorschläge und Löschung ungültiger Zuordnungen entsprächen der (einzigen) „Produktionsregel“ der inkrementellen Transformation. Insofern stellt sich die Frage, ob nicht gleich das Konzept der inkrementellen Transformation wiederverwendet werden kann. Die entscheidene Lücke betrifft dabei den Übergang von Zuordnungsvorschlägen zu geltenden Zuordnungen. Dieser Übergang wird von einer Person vorgenommen, die nur einige der Zuordnungsvorschläge annimmt. Eine inkrementelle Transformation aber reduziert immer alle gefundenen Redexmorphismen.

Inkrementelle Transformation mit Vorschlägen. Die Lösung zur Anwendung der inkrementellen Transformation für die Unterstützung der Kontrolle besteht darin, diese um Benutzerinteraktion zu erweitern. Konkret wird die Redexauswahlfunktion so ausgelegt, dass sie eine externe Quelle einbezieht, um die Entscheidung zu treffen, welche der möglichen Redexmorphismen tatsächlich angewendet werden. Die möglichen Redexmorphismen werden somit als Vorschläge betrachtet (die dem Projektleiter z.B. in optisch in aufbereiteter Form präsentiert werden können).

Einmal manuell getroffene Entscheidungen (Zuordnungen) sollen nicht bei jeder weiteren inkrementellen Transformationsanwendung erneut manuell getroffen werden. Folglich müssen diese persistiert werden. Im Bezug auf das Szenario der inkrementellen Transformation (Kapitel 2.3.1) kann dies jedoch nicht im Graph g1 (bzw. in dessen um manuelle hinzugefügte Informationen erweiterte Fassung g1') erfolgen: Die gesetzten

[37] Beispielsweise nach dem Einchecken der Version lv34 in das zentrale Repository, und noch bevor der Prüfer sich aus dem Repository die vorgelegte Version der Systemarchitektur ausgecheckt hat.

Zuordnungen werden für die Anwendung der Planungsvorgaben ausgewertet. Beispielsweise ist für die Soll-Version-Planung relevant, ob und welche Ist-Version eines Ist-Ergebnisses in welchem Zustand und mit welchen Ist-Inhalten vorhanden ist. Daher müsste g1 (bzw. g1') von der inkrementellen Transformation als Eingabe verarbeitet werden. Andererseits können Zuordnungen auch nicht im Graph g0 (der tatsächlich als Eingabe verwendet wird) abgelegt werden, da sie sich auf Soll-Versionen beziehen, die durch die inkrementelle Transformation erst erzeugt werden. Folglich lässt sich das Szenario der inkrementellen Transformation so nicht anwenden.

Induktiv inkrementelle Transformation mit Vorschlägen. Die Lösung zur Anwendung der inkrementellen Transformation mit Vorschlägen (siehe vorangehender Paragraph) besteht darin, das Szenario der inkrementellen Transformation wie folgt anzupassen: Anstelle der Betrachtung zweier Graphen g0 und g1, wird insgesamt nur ein Graph g betrachtet. Die Ausgabe einer inkrementellen Transformationsanwendung wird die Eingabe für die jeweils nächste Transformationsanwendung. Damit können zwischenzeitlich vorgenommene Änderungen sowohl persistiert als auch als Eingabe für die inkrementelle Transformation verarbeitet werden.

5.1.3 Automatisierung der Steuerung

Naive Implementierung. Analog zur Automatisierung der Planung liegt auch zur Automatisierung der Steuerung der Lösungsansatz nahe, die in der Aufgabenstellung gesuchte Funktion steuerung einfach in Form eines SW-Programms zu implementieren. Doch auch hier darf eine solche Implementierung nicht beliebig sein, was im Folgenden an einem Beispiel erläutert wird. Dazu seien die in Abbildung 5.11 gezeigten Produktionsregeln s1 und p1 betrachtet. Die Produktionsregel p1 stellt hier die Automatisierung der Planung dar, bei der zu jedem initialen und als relevant befundenen Ergebnistyp ein Soll-Ergebnis eingeplant wird. Die Produktionsregel s1 stellt die Automatisierung der Steuerung als naive Implementierung in Form einer einfachen (also nicht-inkrementellen) Transformation dar. Konkret wird der Ergebnistyp Lastenheft als relevant markiert, falls ein AG-Projekt durchgeführt wird.

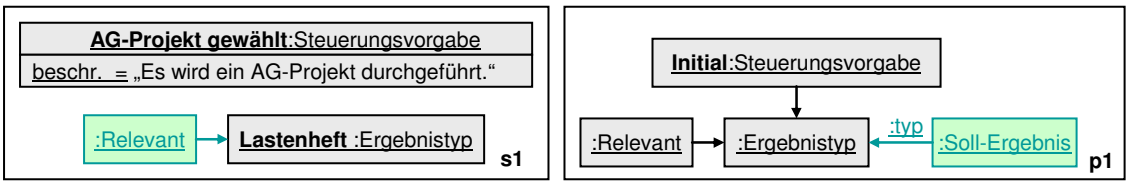


Abbildung 5.11 Zusammenhänge Steuerungs- und Planungsvorgaben am Beispiel

Abbildung 5.12 zeigt, dass diese naive Implementierung der Steuerung ebenfalls, diesmal mittelbar, zum Verlust von Zuordnungen zwischen Soll und Ist führt. Dies resultiert daraus, dass die Produktionsregeln für die Planung sich auf Elemente beziehen (d.h. diese in die Transformationshistorie aufnehmen) die von der Automatisierung der Steuerung erzeugt werden.

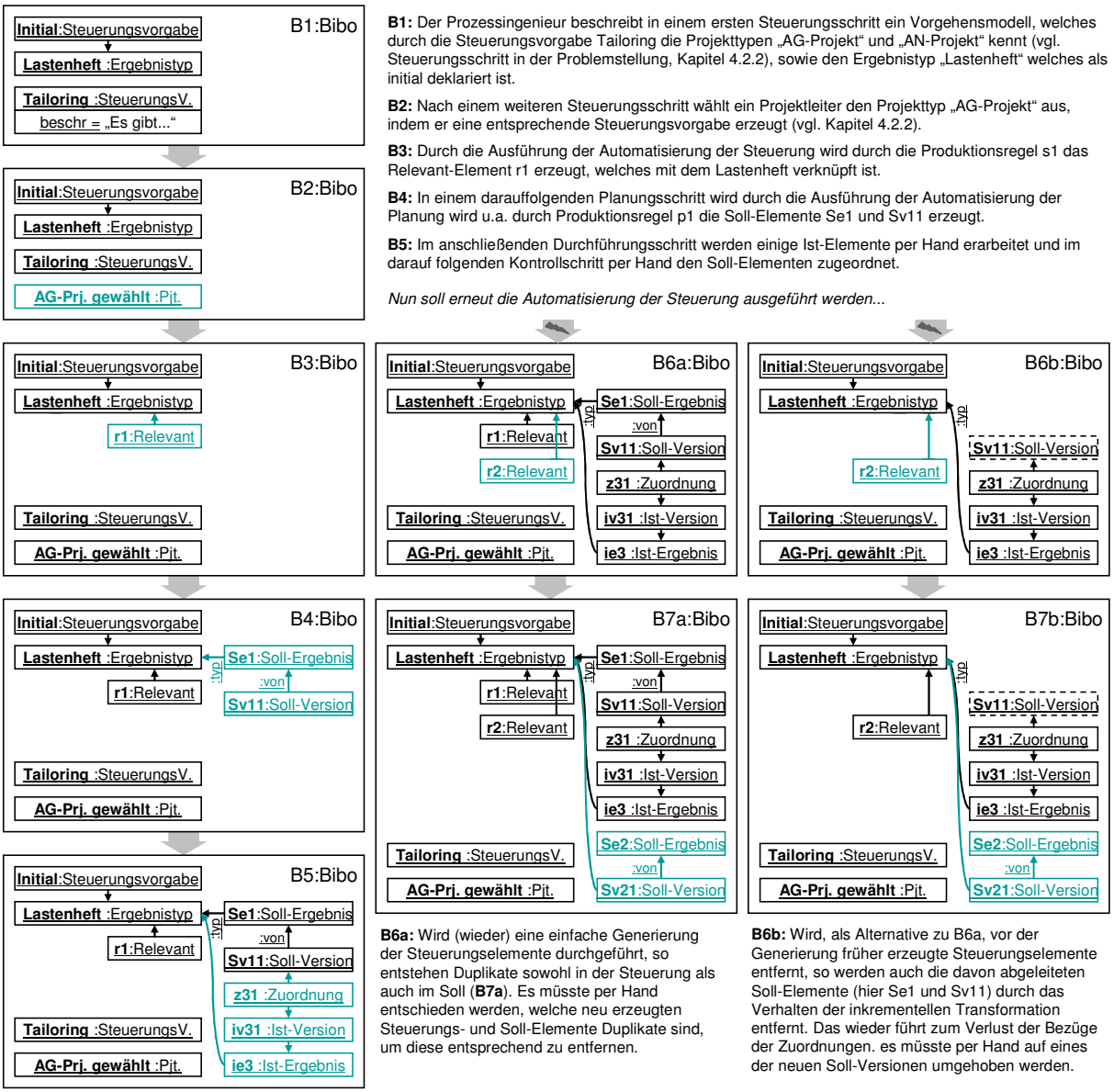


Abbildung 5.12 Herausforderung bei der Automatisierung der Steuerung

Inkrementelle Transformation. In Analogie zur Planung wird das Werkzeug der inkrementellen Transformation auch zur Automatisierung der Steuerung verwendet. Dazu werden (formalisierbare) Steuerungsvorgaben ebenfalls als virtualisierte Produktionsregeln in einer Produktbibliothek abgelegt.

Varianz der Ursachen, Konstanz der Folgen. Einige Steuerungsvorgaben sind nicht pro Element mit bestimmten Eigenschaften anzuwenden, sondern nur einmal wenn eine bestimmte Situation vorherrscht. Als Beispiel sei die in Abbildung 5.13 durch eine Produktionsregel formalisierte Steuerungsvorgabe betrachtet, die den Ergebnistyp SW-Spezifikation als relevant markiert, sobald und solange eine beliebige SW-Einheit identifiziert ist.

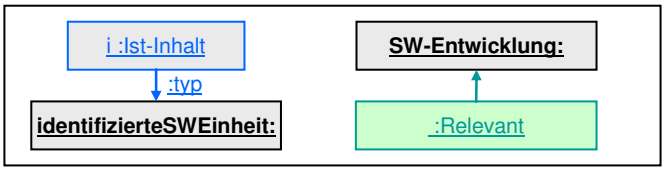


Abbildung 5.13 Beispiel zur Nutzung der Ignorierung für das Projektcontrolling

Für die Automatisierung solcher Steuerungsvorgaben ist es notwendig, auswählen zu können, welche Elemente einer Belegung in die Transformationshistorie aufgenommenen werden. In diesem Beispiel ist insbesondere die Belegung des Suchknotens i nicht in die Transformationshistorie aufzunehmen. Zur Erläuterung sei der Projekthergang betrachtet, bei dem zum Zeitpunkt t1 eine SW-Einheit e1 identifiziert und bis zum Zeitpunkt t2 durch die SW-Einheiten e2 ersetzt wird. Bei der inkrementellen Transformation zum Zeitpunkt t2 werden alle Soll-Elemente die zum Zeitpunkt t1 automatisch erstellt wurden, und auf der Relevanz der SW-Spezifikation beruhen, zunächst entfernt (da die entsprechenden Regelmorphismen mangels e1 nicht mehr bildbar sind), und anschließend neue Soll-Element erzeugt, da die Relevanz der SW-Spezifikation durch e2 wiederhergestellt wird. Problematisch ist die Neuerstellung der Soll-Elemente sofern diese referenziert werden – wie etwa von manuell erstellten Ist-Soll-Zuordnungen oder weiteren Daten die an die Soll-Elemente geknüpft wurden (wie bspw. eingeplante Personen zu deren Erledigung).

Mit einer normalen Produktionsregel lässt sich jedoch nicht steuern, was in die Transformationshistorie aufgenommen wird – der gesamte Redexmorphismus wird abgelegt. Zwar erlaubt die inkrementelle Transformation mit Tripel-Graph-Grammatiken, technisch gesehen, die Einschränkung des aufgenommenen Redexmorphismus. Dazu allerdings sind Domänen notwendig, die hier, je nach Produktionsregel, anders liegen müssten. Das würde insbesondere bedeuten, dass auch die Domänenzuordnung in der zu transformierenden Produktbibliothek vor jeder Produktionsregelanwendung passend zurechtgeschoben werden müsste. Die Lösung besteht in der Nutzung derim Kapitel 2.4.3 eingeführte Konzept der Ignorierung. Mit dieser kann gezielt gewählt werden, die Belegungen welcher Musterknoten in den Transformationshistorie aufzunehmen sind.

5.1.4 Integrationsmethodik für Prozessbeschreibungssprachen

Während aus Sicht eines Planungsschritts die Vorgaben eines Vorgehensmodells nur anzuwenden sind, können sie aus Sicht eines Steuerungsschritts auch zu definieren bzw. auszugestalten sein. Der für diese fachliche Aufgabe verantwortliche Prozessingenieur ist in der Praxis eine Person, die typischerweise nicht mit Programmiertätigkeiten beschäftigt ist. Dies zeigt sich insbesondere daran, dass für die Definition von Vorgehensmodellen gerade nicht turingmächtige Programmiersprachen wie Java, sondern die viel „einfacheren“ Prozessbeschreibungssprachen (vgl. Kapitel 3) eingesetzt werden. Das Werkzeug der inkrementellen Transformation, basierend auf Graphen und Produktionsregeln, ist vom Abstraktionsgrad jedoch mit Programmiersprachen vergleichbar (sofern für komplexere Bedingungen OCL verwendet wird[38]). Eine Automatisierung der Steuerung wäre damit für die Praxis nicht zumutbar.

Integration Prozessbeschreibungssprache und Automatisierungskonzept. Die Lösung besteht darin, eine Prozessbeschreibungssprache mit dem Automatisierungskonzept zu integrieren. Dazu sind einerseits Konstrukte hinzuzufügen, um die noch nicht formalisierten Teile wie die Planungs- und Strukturvorgaben zu formalisieren, und andererseits die Semantik so umzugestalten, dass sie auf Graphen und Produktionsregeln abbildet. Um nicht auf eine konkrete Prozessbeschreibungssprache beschränkt zu sein, wird die Lösung als Integrationsmethodik aufgestellt. Diese nimmt eine Prozessbeschreibungssprache aus der Praxis als Eingabe, und liefert dessen mit dem Automatisierungskonzept integrierte Variante samt zugehöriger Semantik (*Semantikabbildung*) als Ausgabe. Als vereinfachende Grundlage zur Definition der Semantik wird das Aufstellen einer *Verfeinerungsabbildung* angeboten, mit der das Produktbibliotheksmodell verfeinert werden kann. So können spezifische Konstrukte

[38] Sofern OCL nicht verwendet wird, wäre der Abstraktionsgrad unterhalb von Programmiersprachen anzusetzen, da Fallunterscheidungen, rekursive Navigation durch Graphelemente etc. mit mehreren Produktionsregeln nachgebildet werden müssten.

der zu integrierenden Prozessbeschreibungssprache direkt übernommen werden. Abbildung Abbildung 5.14 zeigt die Anwendung der Integrationsmethodik an einem konkreten Beispiel.

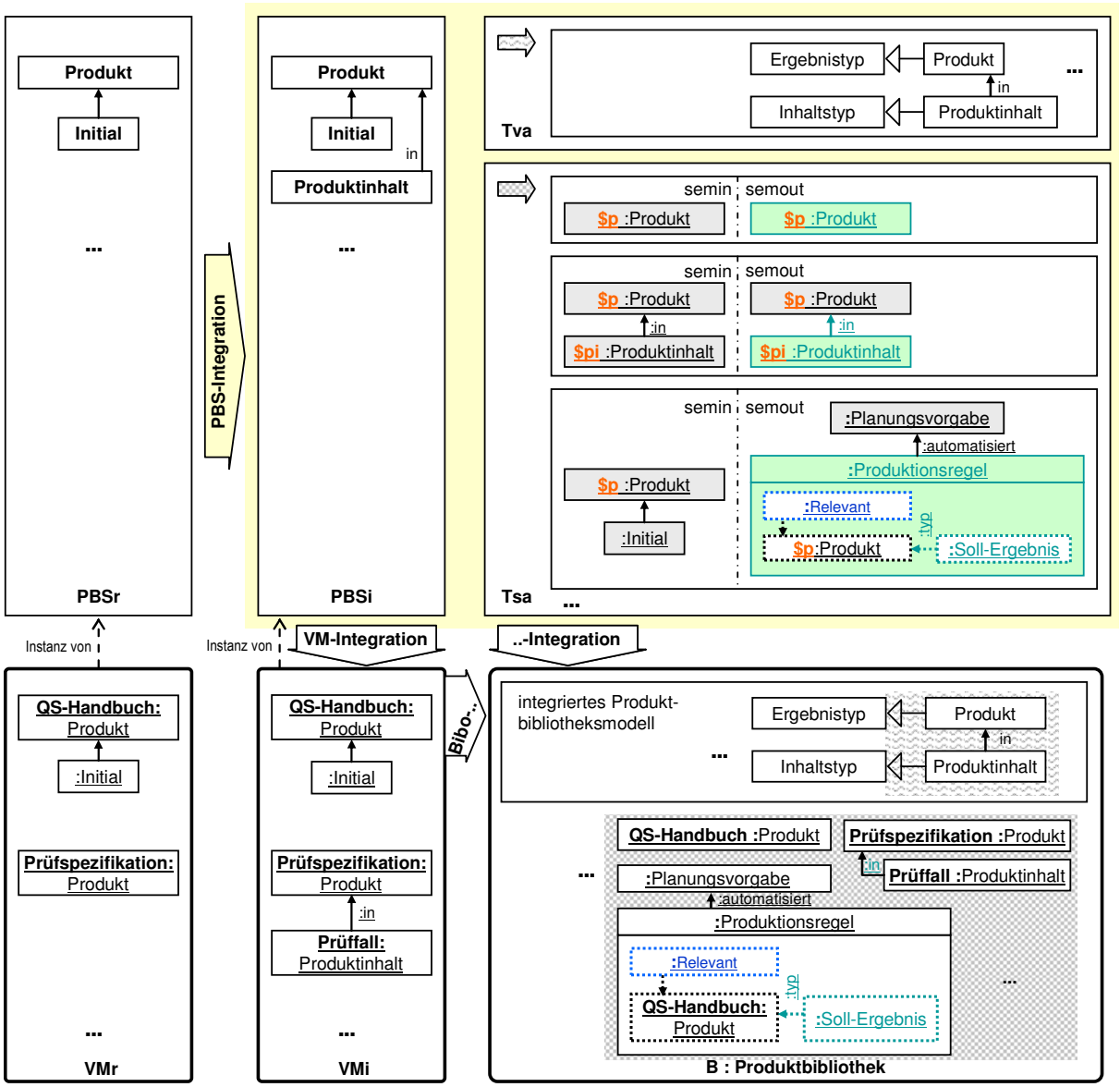


Abbildung 5.14 Beispiel einer Anwendung der Integrationsmethodik

Der erste Schritt der Integrationsmethodik ist die Prozessbeschreibungssprachen-Integration (PBS-Integration), die als Eingabe die zu integrierende Prozessbeschreibungssprache PBSr nimmt. Nach deren Anwendung liegen die integrierte Prozessbeschreibungssprache PBSi, die Verfeinerungsabbildung Tva sowie die Semantikabbildung Tsa vor. Die integrierte Prozessbeschreibungssprache PBSi wurde hier um das Konstrukt Produktinhalt erweitert – z.B. weil dies als hilfreich erachtet wurde um bestimmte Vorgaben formalisieren zu können. Die Verfeinerungsabbildung Tsa verfeinert die Konstrukte Ergebnistyp bzw. Inhaltstyp der Produktbibliotheksstruktur um die Subtypen Produkt und Produktinhalt. Eine Verfeinerung ist dabei nicht nur auf Subtypen beschränkt, sondern kann auch neue Kanten umfassen, wie dies am Beispiel von in zu sehen ist.

Der zweite Schritt der Integrationsmethodik ist die Vorgehensmodell-Integration (VM-Integration), nach deren Anwendung das integrierte Vorgehensmodell VMi vorliegt. Dieses wurde gemäß der bereits integrierten Prozessbeschreibungssprache PBSi angepasst. Die VM-

Integration setzt jedoch nicht voraus, dass es zuvor ein nicht integriertes VMr überhaupt gibt – es kann auch ein komplett neues Vorgehensmodell in diesem Schritt erstellt werden.

Der dritte Schritt in der Integrationsmethodik ist die Bibo-Integration, nach deren Anwendung eine im Rahmen des Automatisierungskonzepts sofort nutzbare Produktbibliothek vorliegt. Dazu wird die initiale Produktbibliothek genommen, und diese mit der Verfeinerungsabbildung Tva sowie der Semantikabbildung Tsa modifiziert. Die Anwendung der Verfeinerungsabbildung führt zur Verfeinerung des Produktbibliotheksmodells. Die Anwendung der Semantikabbildung, unter Einbeziehung des im vorherigen Schritt integrierten Vorgehensmodells, führt zu einer Befüllung der Teilbibliothek Strg.

Intuitive Formalisierung. Eine Reihe von Vorgehensmodellinhalten können mit der zu integrierenden Prozessbeschreibungssprache nur informell vorgegeben werden. Sofern sich diese, wie für das Beispiel von VM_0 die erzeugenden Abhängigkeiten, auf die Schritte Planung und Steuerung beziehen, müssen sie jedoch im Rahmen der Integration mit dem Automatisierungskonzept durch Konstrukte mit formaler Semantik hinterlegt werden. Hierzu wird die Technik der *intuitiven Formalisierung* aufgestellt. Dabei wird die Erarbeitung der fehlenden Teile der Prozessbeschreibungssprache als ein SW-Entwicklungsprojekt eines Informationssystems angesehen: Die Sprachkonstrukte („Schnittstelle“ des Systems) mit der darunter liegenden Semantik („Implementierung“ des Systems) werden aus den zu erwartenden Forderungen der Planungs- und Strukturvorgaben („Use Cases“) abgeleitet. Der Prozessingenieur („Anwender“ des Systems) bekommt somit nur diejenigen Sprachkonstrukte zur Verfügung, die für seine fachliche Aufgabe adäquat sind. Im Vergleich zu einem Informationssystem hat die Schnittstelle allerdings einen technischen Unterschied: Sie besteht darin, dass der Prozessingenieur strukturierte Texte anhand eines vorgegebenen Schemas ableitet – und nicht etwa die Konstrukte im Sinne einer interaktiven Oberfläche aufruft. Dieses Schema wird zunächst als ein UML-Aktivitätsdiagramm erarbeitet. Abbildung 5.15 zeigt dazu ein Beispiel.

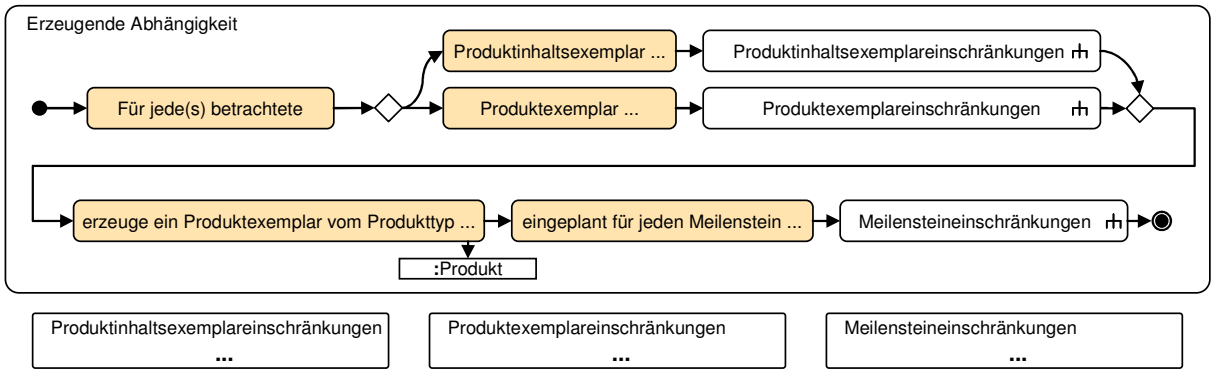


Abbildung 5.15 Beispiel für UML-Aktivitätsdiagramme als Sprachdefinition

Jeder Pfad durch ein Aktivitätsdiagramm repräsentiert ein gültiges Wort der erweiterten Prozessbeschreibungssprache. Dabei bilden normale Aktionen mit ihrem Namen Terminale, Callbehavior-Aktionen detaillieren das jeweils unmittelbar vorherige Terminal durch ein untergeordnetes Wort, und Objektknoten repräsentieren normale Datenkonstrukte der erweiterten Prozessbeschreibungssprache, wodurch Terminale darauf Bezug nehmen können.

Da ein Vorgehensmodell ein Graph und kein Pfad eines UML-Aktivitätsdiagramms ist, wird in einem zweiten Schritt ein Modell aus den UML-Aktivitätsdiagrammen abgeleitet, welches die zu integrierende Prozessbeschreibungssprache erweitert. Abbildung 5.16 zeigt, wie das Resultat für das Beispiel aus Abbildung 5.15 konkret aussieht.

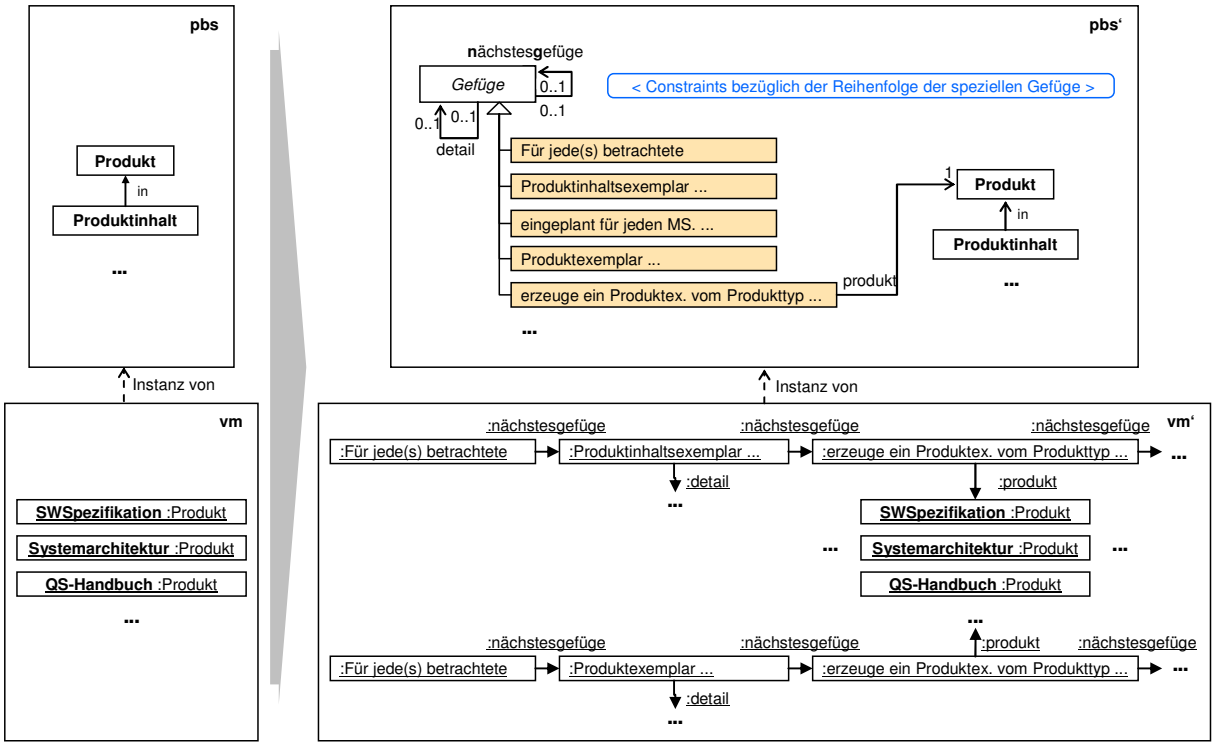


Abbildung 5.16 Ausgestaltung eines Vorgehensmodells mit Gefügeketten

Pfade werden durch Gefügeketten dargestellt, wobei jedes Gefüge genau ein Terminal (durch den jeweils spezielleren Typ) repräsentiert und höchstens ein nächstes Gefüge hat. Dadurch entstehen lesbare, intuitiv verständliche Texte – sofern im Rahmen der Erstellung der Aktivitätsdiagramme passende Terminale gewählt wurden. Um das Verständnis durch Details nicht zu erschweren, sind diese Gefügeketten hierarchisch (Kante mit Typ detail), so dass jede Hierarchiestufe für sich gelesen werden kann. Im Rahmen natürlicher Sprachen können Hierarchien als „Text in Klammern“ verstanden werden. In welcher Reihenfolge die Gefüge verbunden werden dürfen, wird formal durch Constraints festgehalten, die aus den Verbindungen des UML-Aktivitätsdiagramms abgeleitet werden. Diese Constraints sind jedoch nur für eine technisch fundierte Sicherstellung der Anwendung der UML-Aktivitätsdiagramme gedacht – nicht als Anleitung für einen Prozessingenieur. Anders formuliert dienen die aufgestellten UML-Aktivitätsdiagramme dem Prozessingeneiuer beim erstellen eines Vorgehensmodells auch als Anleitung für das Aufstellen von Gefügeketten. Das Modell der Gefügeketten wird direkt in die Prozessbeschreibungssprache eingefügt. In Abbildung 5.16 entsteht so aus pbs die Prozessbeschreibungssprache pbs'. Auf Ebene der damit definierten Vorgehensmodelle wird aus vm entsprechend vm'. Die Verbindung zwischen den Gefügeketten und den anderen Konstrukten aus dem bisherigen Stand der Prozessbeschreibungssprache, pbs, wird durch Assoziationen realisiert, die aus den Objektknoten abgeleitet werden. In diesem Fall betrifft dies die Auswahl des Produkts, für das unterste Terminal der Abbildung.

Abschließend wird, im Rahmen der Definition der Semantikabbildung, die gesamte Sprache mit einer Semantik unterlegt. Für die Gefügeketten wird dabei je Anwendungsfall eine Produktionsregel-Schablone verwendet. Diese Schablone wird mit Elementen befüllt, die sich aus dem „Zusammenrechnen“ der Ketten ergeben. Anders formuliert wird jede Gefügekette einschließlich aller Hierarchien durch die Semantikabbildung auf genau eine Produktionsregel reduziert, die zur Teilbibliothek Strg hinzugefügt wird.

5.1.5 Zusammenfassung

Abbildung 5.17 fasst das Lösungskonzept zusammen. Für die Automatisierung der Planung und Steuerung wird das Produktbibliotheksmodell hinsichtlich des Teilmodells Strg erweitert[39]. Dieses enthält virtualisierte Produktionsregeln, die im jeweiligen Schritt des Projektcontrollings durch eine inkrementelle Transformation ausgewertet werden. Für die Planung wird das Teilmodell Strg so angepasst, dass Zustandsübergänge als optimistisch markiert werden können. Anhand dieser Markierungen wird die Erzeugung von Soll-Versionen so gestaltet, dass diese einerseits weit im Voraus eingeplant werden, andererseits bei Abweichungen vom optimistischen Verlauf automatisch durch die inkrementelle Transformation zurückgebaut werden. Für die Unterstützung der Kontrolle wird der Teil SollIstVgl des Produktbibliotheksmodells so erweitert, dass Ist-Versionen vom Projektleiter von der Erstellung von Zuordnungsvorschlägen ausgeschlossen werden können. Anhand der Vorschläge erstellt der Projektleiter schließlich die (von der Automatisierung der Steuerung und Planung verwendeten) Zuordnungen. Da Steuerungs- und Planungsvorgaben in der Praxis oft organisations- oder projektspezifisch auszugestalten sind, müssen diese von dem dafür zuständigen Prozessingenieur formuliert bzw. vom Projektleiter ausgestaltet werden. Um dies nicht auf der Ebene von Graphen und Produktionsregeln zu belassen, wird eine Integrationsmethodik für Prozessbeschreibungssprachen angeboten. Diese erweitert eine existierende Prozessbeschreibungssprache um für das Automatisierungskonzept notwendige Konstrukte und einer auf Graphen und Produktionsregeln basierenden Semantik – so dass definierte Vorgaben inkrementell angewendet werden können.

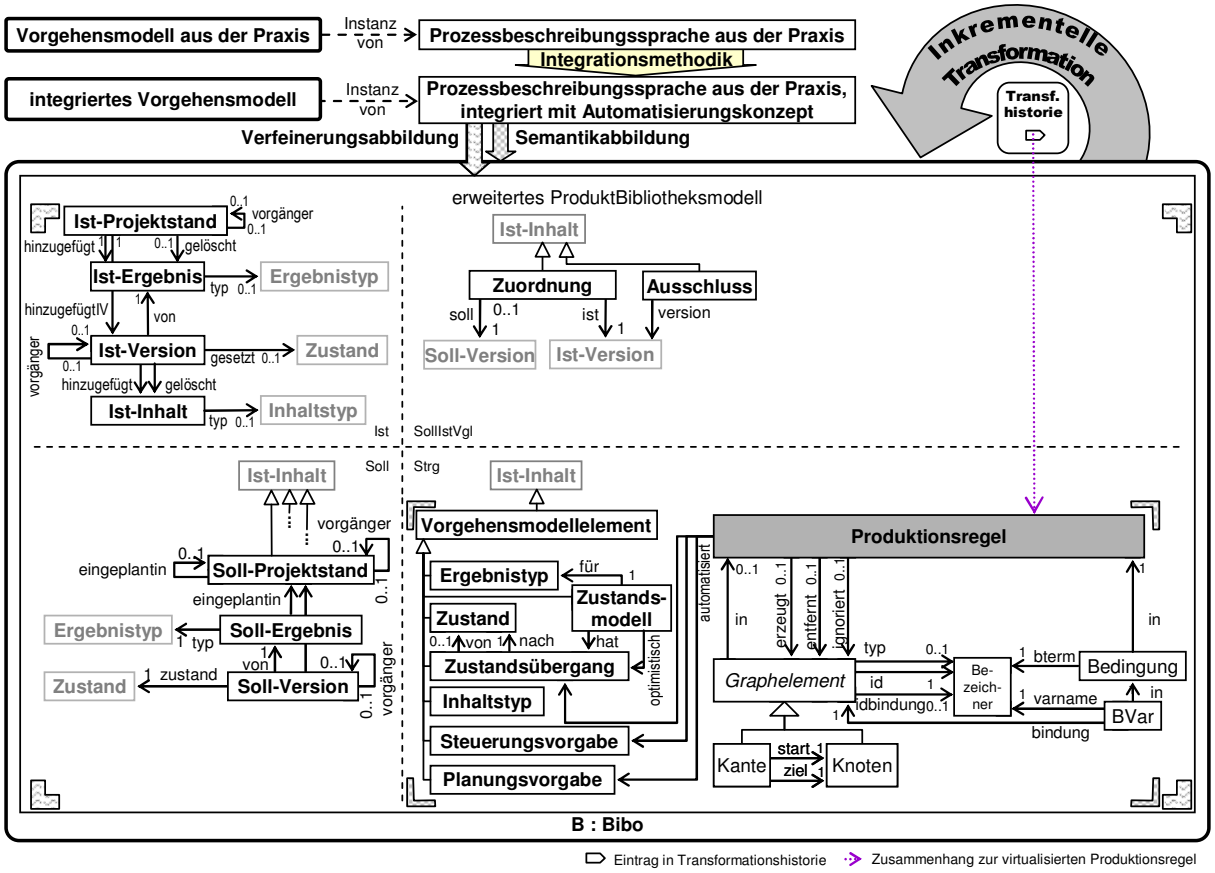


Abbildung 5.17 Zusammenfassung Lösungsansatz

[39] Für ein formal korrektes Modell sind die drei Assoziation vom Typ automatisiert durch eine einzige zu ersetzen, die eine Klasse referenziert, von der nur die Konstrukte Zustandsübergang, Steuerungs- und Planungsvorgabe erben. Zur Übersichtlichkeit wird im Folgenden darauf verzichtet.

5.2 Erweiterungen für Graphen und Transformationen

Die erweiternden Konzepte werden in gleicher Weise wie in Kapitel 2 – als Differenz zu den Grundlagen – eingeführt. Da diese allgemein sind, werden sie unabhängig vom Projektcontrolling dargestellt.

5.2.1 Inkrementelle Transformation mit Constraints

Das Formulieren von Produktionsregeln die nur dann greifen, wenn im zu transformierenden Graphen ein komplexer Sachverhalt vorliegt (z.B. eine zyklische Struktur beliebiger Länge), kann schnell sehr aufwendig werden. So müssen gegebenenfalls Hilfs-Produktionsregeln definiert werden um erst durch ihr abgestimmtes Zusammenspiel den gewünschten Effekt zu erreichen. Die erhöhte Zahl von Produktionsregeln ist jedoch schwerer zu verwalten, so dass kompaktere Ausdrucksmöglichkeiten für das Formulieren komplexer Bedingungen nötig sind.

Die Verwendung von Constraints bietet die Möglichkeit, das Formulieren von Bedingungen unabhängig von der Struktur der Produktionsregeln zu gestalten. Aus Sicht einer Produktionsregel erscheint ein Constraint als eine Blackbox, die zur ihr hinzugefügt werden kann, und im Rahmen einer (inkrementellen) Transformation die anstehenden Reduktionen auf ihre Zulässigkeit begutachtet. Analog zu den Constraints bei Modellen (Kapitel 2.6.1) bleibt freigestellt, wie die interne Struktur der Constraints organisiert wird. Ein Constraint muss also nicht wie die Produktionsregel durch Graphen formuliert werden, sondern kann auch eine textuelle Sprache sein. Entsprechend wird auch hier OCL eingesetzt.

Abbildung 5.18 zeigt, wie ein OCL-Constraint als Teil einer Produktionsregel dargestellt wird. Ein Constraint ist jedoch kein Graphelement, und ist weder Teil des zu transformierenden Graphen g, noch Teil des resultierenden Graphen g'. Konkret besagt der Constraint in p1, dass p1 nur dann Angewendet werden kann, wenn kein Knoten vom Typ Y1 alle Knoten vom Typ Y2 mit einer Kante vom Typ a referenziert. Da n1 sowohl n3 als auch n4 referenziert, ist g'=g.

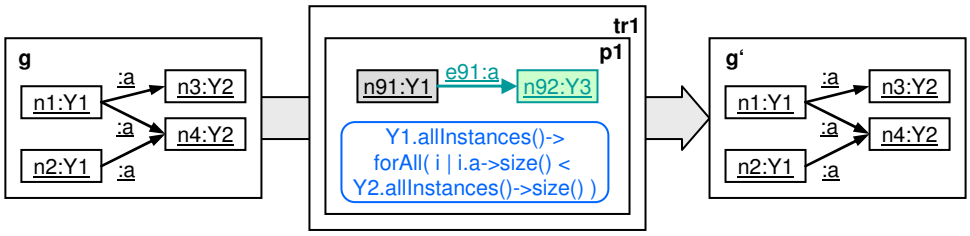


Abbildung 5.18 Notation und Wirkung eines (OCL-)Constraints

In dieser Arbeit wird unter einem Kontext eines Constraints die Bindung dessen freier Variablen verstanden. Der Constraint in Abbildung 5.18 hat keine freien Variablen und benötigt daher nicht mehr als eine leere Bindung. Der Constraint in Abbildung 5.19 hat die freie Variable x. Zur Auswertung des Constraintterms muss x jedoch mit einem konkreten Graphelement belegt werden. Entsprechend wird eine Bindung dieser Variable benötigt. Die mit x beschriftete Kante vom Constraint zum Suchmusterknoten n91 stellt eine solche Bindung dar. Damit können Produktionsregeln nicht nur für alle oder keine Redexmorphismen als anwendbar definiert werden, sondern auch in Abhängigkeit der konkreten Belegungen der Suchmusterknoten: Ein Redexmorphismus hx wird nur dann (bezüglich des Constraints) als nicht anwendbar erachtet, wenn er, in diesem Beispiel, unter Einsetzen von hx(n91) für x nicht erfüllt ist. Der Redexmorphismus, der n91 auf n1 abbildet, ist so ein Fall. Für den Redexmorphismus, der n91 auf n2 abbildet – und damit auch n2 für x eingesetzt wird – ist der Constraintterm erfüllt, weshalb der Knoten n5 in g'' erzeugt an entsprechender Stelle erzeugt wird.

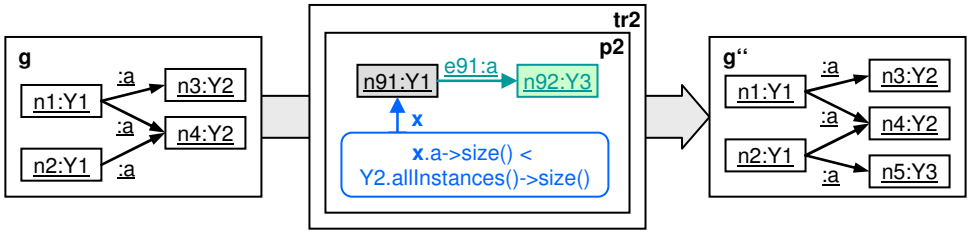


Abbildung 5.19 Notation und Wirkung eines (OCL-)Constraint mit freien Variablen

OCL-Constraints haben jedoch keine freien Variablen. Sämtliche Variablen müssen an eine Iteration gebunden werden. Einzige Ausnahme bildet dabei die Variable self, die durch einen OCL-Kontext an einen Typ gebunden wird. Ein OCL-Constraint kann nur dann insgesamt erfüllt sein, wenn er für jede Instanz dieses Typs erfüllt ist. Das Beispiel aus Abbildung 5.18 liesse sich damit wie folgt als OCL-Constraint mit OCL-Kontext darstellen:

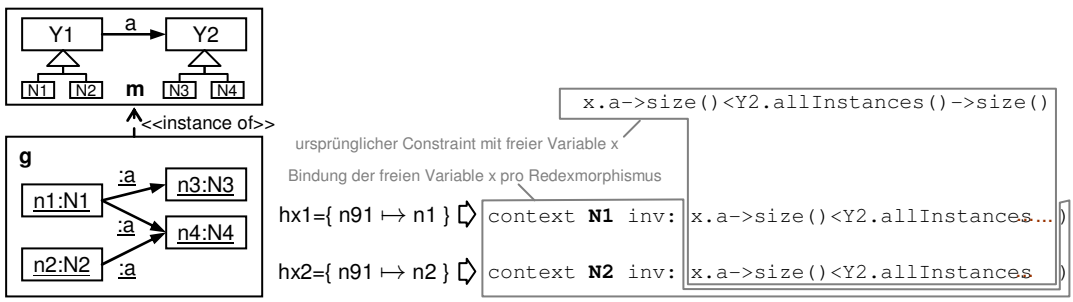
```
context Y1 inv: self.a->size() < Y2.allInstances()->size()
```

Das Beispiel aus Abbildung 5.19 lässt sich in OCL jedoch nicht darstellen, da OCL-Constraints nicht für die Sondierung von Redexmorphismen bezüglich einer Produktionsregel ausgelegt wurden, sondern für die Sondierung von Graphen bezüglich eines Modells (siehe Kapitel 2.6.1). Konkret äußert sich dies dadurch, dass es in OCL keine Konstrukte gibt, mit denen für einen zu sondierenden Redexmorphismus auf dessen Belegung eines Suchmusterknotens zugegriffen werden kann. So funktioniert der folgende, „intuitive“ Ansatz *nicht*, da `n91` in `g` kein Typ ist:

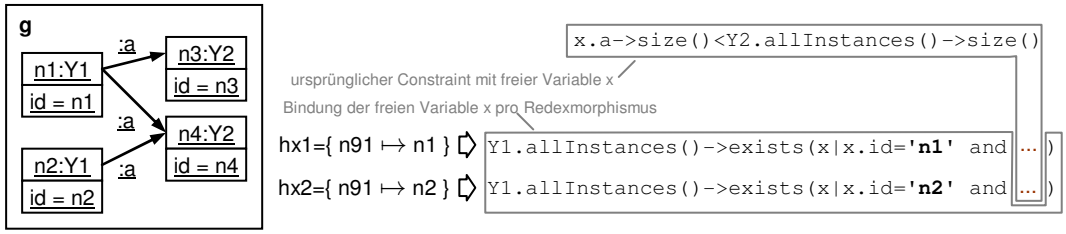
```
context n91 inv: self.a->size() < Y2.allInstances()->size()
```

Die Variable `n91` in diesem OCL-Constraint wird also gerade nicht, wie intuitiv vermutbar, für jeden zu sondierenden Redexmorphismus mit dessen Belegung des Suchmusterknotens `n91` belegt. Prinzipiell gibt es folgende Möglichkeiten dies dennoch zu erreichen:

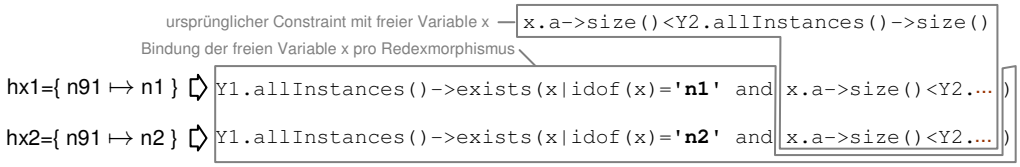
- **Singleton-Typ:** Es wird vereinbart, dass jeder Knoten einen eigenen Typ hat („A class of it’s own“), zu dessen Typ es also keinen anderen Knoten als Instanz gibt. Solche Typen werden als Singleton-Typen bezeichnet. Die ursprünglichen Typbeziehungen werden dadurch erhalten, indem diese auf die Singleton-Typen übertragen werden. Ein Singleton-Typ erbt dabei von dem Typ, der dessen Instanz ursprünglich hatte. Bei der Zur Auswertung des Constraints wird die freie Variable `x` durch einen OCL-Kontext *pro* Redexmorphismus an die Belegung des Suchmusterknotens `n91` gebunden:



- **Id-Attribut:** Es wird vereinbart, dass jeder Knoten (der keinen Primitvwert darstellt, vgl. 2.4.6) einen Slot (vgl. Abbildung 2.50 im Kapitel 2.6.6) vom Typ `id` hat, dessen Wert der Id des Knotens entspricht, und als Zeichenkette vorliegt. Unter der Zusicherung dass jede Id höchstens einmal vorkommt, ist die Bindung der Variable `x` wie folgt *pro* Redexmorphismus ausdrückbar:



- **Id-Operator:** Zunächst wird OCL um den Operator `idof` erweitert, mit dem die Id eines Objekts als eine Zeichenkette bezogen werden kann. Zur Auswertung des Constraints wird die freie Variable x *pro* Redexmorphismus wie folgt gebunden:



Die erste Möglichkeit skalliert nicht für mehrere freie Variablen, da in OCL immer nur ein Kontext angegeben werden kann. Die zweite Möglichkeite erfordert (wie auch die erste) eine geeignete Verwaltung der Graphen, durch die Id-Attribute – z.B. im Rahmen einer inkrementellen Transformation neu erzeugter Knoten – automatisch gesetzt werden. Die dritte Möglichkeit erfordert einzig die Anpassung von OCL. Da OCL nur als Beispiel einer Constraint-Sprache gewählt wurde, wird deren Anpassung für das Weitere gewählt. So wirken sich Beschränkungen der gewählten Constraint-Sprache nicht auf das Grundlegende Konzept aus.

In [43] wurde bereits eine Integration von OCL mit einfachen Transformationen vorgestellt. Produktionsregeln einer Transformation wurden als Operationen einer UML-Klasse dargestellt, wobei die Such- und Ersetzungsmuster in Pre- und Post-Conditions mit globalen[40] Kontext übersetzt wurden. Die Berechnung des Transformationsergebnisses ergibt sich so aus dem erschöpfenden Anwenden dieser Operationen. Es wird jedoch nicht darauf eingegangen, wie dieses erschöpfende Anwenden konkret aussieht, und ob und wodurch es überhaupt terminiert. Denn eine Post-Condition stellt zunächst nur eine Spezifikation für das Ergebnis einer Operationsanwendung dar. Sollen durch eine solche Operation beispielsweise neue Elemente erzeugt werden, so sind allein durch die freie Wahl der Id des neuen Elements unendlich viele Möglichkeiten denkbar. Dies spielt insbesondere für die inkrementelle Transformation eine Rolle (Wiederherstellung „alter“ Ids, siehe Kapitel 2.3.1), auf die der Ansatz ebenfalls nicht eingeht. Diese Fragestellung wird in [92] durch den Ansatz beantwortet, der aus einer Produktionsregel nicht nur Pre- und Post-Conditions, sondern auch sogenannte Commands abgeleitet, mit denen die Änderungen des Graphen vorzunehmen sind. Wie diese jedoch im Rahmen einer inkrementellen Transformation korrekt angewendet werden (vgl. 2.4.2 – Terminierungsproblem), ist nicht ersichtlich. Obgleich eine Implementierung dieses Ansatzes existiert, wird die Frage konzeptionell weder aufgestellt noch beantwortet. Gleiches gilt auch für die Belegung der freien Variablen von OCL-Constraints bei deren Auswertung.

Definition 5.1: Constraintvariable
Die Menge CVAR *aller* ***Constraintvariablen*** *sei wie folgt* spezifiziert*:*

$$CVAR \subseteq_{spec} \text{Menge gültiger OCL-Variablen}$$

[40] Die die Operationen enthaltende UML-Klasse repräsentiert den gesamten zu transformierenden Graph.

Definition 5.2: Constraint mit freien Variablen
Für das Konzept der inkrementellen Transformation mit Constraints ist die Spezifikation der Menge CONSTRAINT *um folgende Bedingung zu erweitern:*

$$\text{CONSTRAINT} \supset_{spec} \text{Menge der OCL-Constraints für Invarianten, mit nicht gebundenen (d.h. freien) Variablen, ohne OCL-Kontext}$$

Zusätzlich ist OCL um den Operator `idof` *zu erweitern, der zu einem Objekt dessen Id als Zeichenkette zurückgibt.*

Definition 5.3: Constraintkontext für Redexmorphismen
Die Menge RXCONSTRAINTCONTEXT *aller* ***Constraintkontexte*** *ist wie folgt definiert:*

$$\text{RXCONSTRAINTCONTEXT} =_{def} \{ \text{fp} : \text{CVAR} \rightarrow \text{GE} \}$$

Für ein gegebenes $\text{fp} \in \text{CONSTRAINTCONTEXT}$ gibt ein $(x,j) \in \text{fp}$ an, dass die Constraintvariable x an das Suchmusterelement j (einer bei der inkrementellen Transformation anzuwendenden Produktionsregel) gebunden ist.

Definition 5.4: Constraint für Redexmorphismen
Die Menge RXCONSTRAINT *aller* ***Constraints für Redexmorphismen*** *ist wie folgt definiert:*

$$\text{RXCONSTRAINT} =_{def} \text{CONSTRAINT} \times \text{RXCONSTRAINTCONTEXT}$$

Ein Constraint für Redexmorphismen $(ct,fp) \in \text{RXCONSTRAINT}$ besteht aus einem Constraint ct und einem Constraintkontext für Redexmorphismen. Alle freien Variablen in ct müssen[41] im Definitionsbereich von fp vorkommen.

Definition 5.5: Produktionsregel mit Constraint für Redexmorphismen
Für das Konzept der inkrementellen Transformation mit Constraints ist die Definition der Menge PRODRULE *wie folgt anzupassen:*

$$\text{PRODRULE}^{\circ} =_{def} \{ p \in \text{PRODRULE} \mid \underline{\text{Cnst}}_p \subseteq \text{RXCONSTRAINT} \}$$

Eine Produktionsregel hat den zusätzlichen Bestandteil $\underline{\text{Cnst}}$, der eine Menge von Constraints für Redexmorphismen enthält. Dabei darf jeder Constraint seine Kontextvariablen nur an die in $\underline{\text{Id}}_p$ vorhandenen Suchmusterelemente binden. Diese Bedingung wird jedoch erst in der Anpassung der Patternmatchingfunktion formalisiert, da sonst die Funktionen searchpattern und replacepattern sowie die Verarbeitung hierarchischer Negationsbereiche unter PRODRULE nicht abgeschlossen wären.

Definition 5.6: Belegung Constraintkontext für Redexmorphismen
Die Menge CONTEXTASSIGN *aller* ***Belegungen von Constraintkontexten*** *für Redexmorphismen ist wie folgt definiert:*

$$\text{CONTEXTASSIGN} =_{def} \{ \text{ap} : \text{CVAR} \rightarrow \text{GE} \}$$

[41] Diese Bedingung wird hier nicht formalisiert, da dies eine umfangreiche Betrachtung der OCL-Syntax erfordern würde.

Für ein gegebenes ap ∈ CONTEXTASSIGN gibt ein (x,i) ∈ ap an, dass die Constraintvariable x an das Garphelement i (des bei der inkrementellen transformation zu transformierenden Graphen) gebunden ist.

Definition 5.7: Binden freier Variablen eines OCL-Constraints an Belegung
Seien ein Constraint ct ∈ CONSTRAINT, *eine Belegung* ap ∈ CONTEXTASSIGN *und ein Graph* g ∈ GRAPH *gegeben. Die* ***Bindung der freien Variablen*** *in* ct *bezüglich* ap *und* g*, notiert als* bindocl(ct,ap,g)*, ist wie folgt definiert:*

bindocl(ct, ∅, g) $=_{def}$ ct
bindocl(ct, {x↦i}∪Ap, g) $=_{def}$
  $\underline{y}_g$(i) · '.allInstances()->exists('·x·'|idof('·x·')='·i·' and '·
  bindocl(ct,Ap,g) ·
  ')'

Da Ids Graphelemente eindeutig identifizieren, kann für die Bindung einer Constraintvariable x eine Existenz-Quantifizierung verwendet werden (Alternativ auch über eine All-Quantifizierung mit Implikation). Die Berechnung ist rekursiv um alle Variablenbindungen ineinander geschachtelt zu erzeugen.

Definition 5.8: Patternmatchingfunktion mit Constraints
Für das Konzept der Constraints ist die Patternmatchingfunktion pmf *wie folgt anzupassen:*

pmf°(p,g) $=_{spec}$ { hx∈pmf(p,g) |
  [∀(ct,fp) ∈ $\underline{Cnst}_p$ mit WB(fp) ⊆ $\underline{Id}_{searchpattern(p)}$: cef(g, oclbind(ct,ap,g))] }
wobei
  ap = $\bigcup$ v∈DB(fp): { v↦hx(fp(v)) }

Aus der Menge möglicher Redexmorphismen werden nur solche Redexmorphismen hx beibehalten, bei denen jeder betrachtete Constraint (ct,fp) vor[42] der Reduktion erfüllt ist. Zur Auswertung der Erfülltheit eines Constraints (ct,fp) werden alle freien Variablen in ct durch oclbind gebunden; Die Belegungen (ap) der Suchmusterknoten von p werden aus dem sondierten Redexmorphismus hx entnommen. Betrachtet werden nur solche Constraints, deren freie Variablen an Graphelemente aus dem Suchmuster von p gebunden sind. Dies ermöglicht es, bei der Negationsauswertung Constraints zu formulieren, die Elemente eines Negationsbereichs mit solchen die außerhalb sind in Bezug zu setzen.

5.2.2 Id-Patternmatching

Bei einer Produktionsregel sind bisher sämtliche Graphelemente des Such- und Ersetzungsmusters von ihrer Wirkung her Variablen. So wird im Beispiel in Abbildung 5.20 der Musterknoten n91 bei der Transformation mit allen Knoten aus g belegt, von denen eine Kante ausführt – so dass am Ende alle Kanten gelöscht sind.

[42] Der Ansatz in [92] basiert zusätzlich auf einer Prüfung nach der Reduktion – im Sinne einer Postcondition. Dieses Verhalten kann hier leicht durch integriert werden, dass zunächst eine „Probereduktion“ ohne Constraint vorgenommen wird, und die Constrainterfüllung anschließend mit dem entstandenen Regelmorphismus (anstelle nur des Redexmorphismus) geprüft wird. Im Erfolgsfall wird die Probereduktion als geltende Reduktion gesetzt. Durch dieses Vorgehen stehen für Constraints lediglich die Belegungen der Erzeugungsmusterelemente als zusätzliche Information zur Verfügung. Diese Art von Informationen wird im Weiteren der Arbeit jedoch nicht benötigt, weshalb nur Preconditions formalisiert werden.

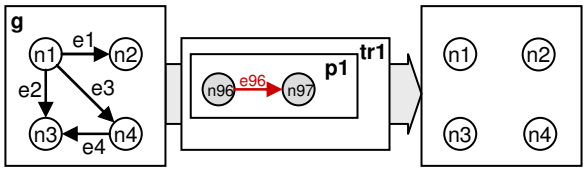


Abbildung 5.20 Patternmatching ohne Beachtung von Ids

Die Menge der als Belegung in Frage kommenden Graphelemente (aus einem zu transformierenden Graphen g) konnte jedoch mit Typen, Constraints und strukturellen Anforderungen an die Umgebung eingeschränkt werden. Eine besondere Form der Einschränkung ist die Einschränkung auf genau ein Graphelement aus g anhand seiner Id. Sofern das gesuchte Graphelement bereits in g vorhanden, und seine Id bekannt ist, kann eine entsprechende Einschränkung durch ein Constraint formuliert werden.

Soll im Rahmen der Transformation das gesuchte Graphelement erst (ggf. optional) erzeugt werden, kann dessen Id zum Zeitpunkt der Definition der Produktionsregeln jedoch nicht bekannt sein. Denn die Ids werden durch die Erzeugungsfunktion auf referenziell nicht transparente Weise vergeben. Um dennoch Produktionsregeln formulieren zu können die auf die Anwesenheit genau eines bestimmten Graphelements aufbauen (oder dieses erzeugen), bestehen prinzipiell folgende Möglichkeiten:

- **Singleton-Typen:** Wie im vorangegangen Kapitel wird für jeden Knoten eines Graphen ein eigener Typ – der Singleton-Typ – eingeführt, der keine anderen Knoten als Instanz haben darf. Sofern die Knoten bereits getypt sind, wird die Typung über eine Vererbungsbeziehung auf den Singleton-Typ übertragen. So kann in einer Produktionsregel auf ein konkretes Graphelement zugegriffen werden (Abbildung 5.21). Bei Verwendung von Singleton-Typen bei Erzeugungsmusterelementen ist die Singleton-Eigenschaft durch ein entsprechendes Constraint (oder Anderweitiges) sicherzustellen.

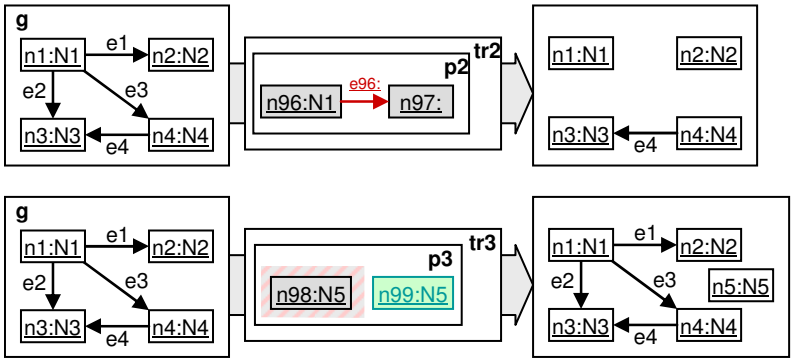


Abbildung 5.21 Id-Patternmatching durch Singleton-Typen

- **Beeinflussung der Erzeugungsfunktion durch Vorgabe der zu vergebenden Id:** Die Produktionsregel-Syntax und deren Auswertungssemantik kann so erweitert werden, dass beim Erzeugen des konkreten Graphelements eine konkrete Id vorgegeben wird. Abbildung 5.22 zeigt wie dies durch ein Bestandteil idm formalisiert werden kann, der als Funktion jedem Musterknoten eine Id zuweisen kann. Bei Suchmusterknoten bedeutet dies die Einschränkung der Belegungsmöglichkeiten für die Patternmatchingfunktion, bei Erzeugungsmusterelementen dagegen eine Vorgabe an die Erzeugungsfunktion. Um Kollisionen vorgegebener Ids mit zufällig erzeugten zu vermeiden, können unterschiedliche Namensräume (z.B. durch einen Präfix) verwendet werden.

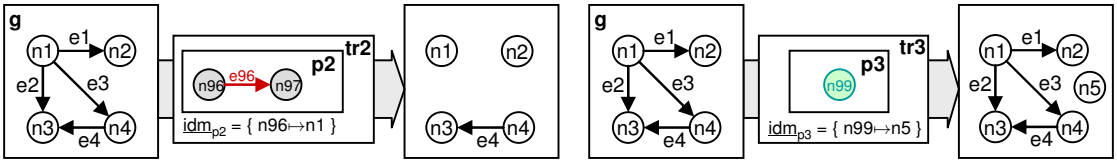


Abbildung 5.22 Id-Patternmatching durch Beeinflussung der Erzeugungsfunktion

Gemäß der bisher verfolgten Systematik für Erweiterungen wird die zweite Möglichkeit verwendet. Zusätzlich sei die in Abbildung 5.23 gezeigte Notation eingeführt.

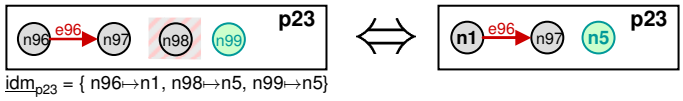


Abbildung 5.23 Notation für Id-Patternmatching

Definition 5.9: Produktionsregel mit Id-Patternmatching
Für das Konzept des Id-Patternmatchings ist die Definition der Menge PRODRULE *wie folgt anzupassen:*

$$\text{PRODRULE}^{\circ} =_{def} \{\, p \in \text{PRODRULE} \mid \underline{idm}_p : \underline{Id}_p \rightarrow GE \,\}$$

Produktionsregeln werden um den Bestandteil $\underline{idm}$ erweitert, welcher als partielle Funktion den Graphelementen aus p eine Id zuweisen kann.

Definition 5.10: Patternmatchingfunktion mit Id-Patternmatching
Für das Konzept des Id-Patternmatchings ist die Spezifikation der Patternmatchingfunktion pmf *wie folgt anzupassen:*

$$pmf^{\circ}(p,g) =_{spec} \{\, hx \in pmf(p,g) \mid \forall\, j \in \underline{Id}_p \cap DB(\underline{idm}_{searchpattern(p)}) : hx(j) = \underline{idm}_p(j) \,\}$$

Jeder durch eine Patternmatchingfunktion ermittelte Graphmorphismus hx muss jedes Musterelement j, welches durch $\underline{idm}_p$ auf eine Id abgebildet wird, auf eben jene abbilden.

Definition 5.11: Erzeugungsfunktion mit Id-Patternmatching
Für das Konzept des Id-Patternmatchings ist die Spezifikation der Erzeugungsfunktion adf *umfolgende Bedingung zu erweitern:*

$$\forall\, p \in \text{PRODRULE}, g \in \text{GRAPH}, hx \in \text{REDEXMORPH}, j \in \underline{Id}_p \cap DB(\underline{idm}_{additionpattern(p)}):$$
$$[rmf(p,g,hx)](j) =_{spec} \underline{idm}_{additionpattern(p)}(j)$$

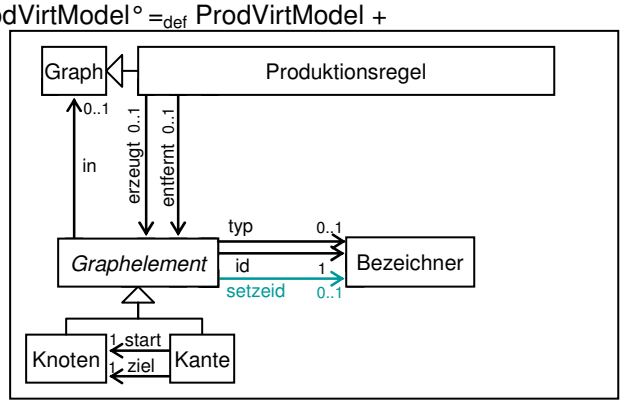
Die Spezifikation von adf wird so erweitert, dass jedes Musterelement j des Erzeugungsmusters von p, welches durch $\underline{idm}_p$ auf eine Id abgebildet wird, auch durch den von rmf zurückgegebenen Ersetzungsmorphismus auf selbige abgebildet wird.

5.2.3 Virtualisierung der Erweiterungen

In diesem Kapitel wird gezeigt, wie die Erweiterung von Produktionsregeln um das Id-Patternmatching und um Constraints in Kombination mit der Dynamischen Transformation (Kapitel 2.7). Konkret sind dazu lediglich die Definitionen für das Virtualisierungsmodell und die De- und Virtualisierungsfunktionen anzupassen.

Definition 5.12: Virtualisierungsmodell für Id-Patternmatching
Für das Konzept des Id-Patternmatchings ist die Definition des Virtualisierungsmodell für Produktionsregeln wie folgt anzupassen (Anpassung grün markiert):

ProdVirtModel° $=_{def}$ ProdVirtModel +



Das Virtualisierungsmodell für Produktionsregeln wird um die Modellkante mit dem Typ setzeid erweitert, dessen Instanzen je einen Eintrag des idm-Bestandteils einer Produktionsregel virtualisieren.

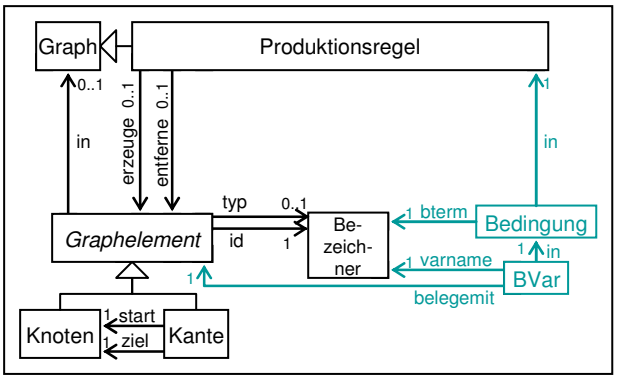
Definition 5.13: Devirtualisierungsfunktion für Id-Patternmatching
Für das Konzept des Id-Patternmatchings ist die Definition der Devirtualisierungsfunktion wie folgt anzupassen:

devirtualizeprod°(g,pid) $=_{def}$ g'[idm$\mapsto$idm'] mit
g' = devirtualizeprod(g,pid),
idm' = $\bigcup$ i $\in$ (instances(Knoten,g)$\cup$instances(Kante,g))$\cap$collectors(pid,in,g) :
[$\bigcup$ d $\in$ collect(i,setzeid,g) : {(d,h(i))}]
wobei
h aus der Berechnung von devirtualizeprod(g,pid) zu entnehmen ist

Die Devirtualisierung eines virtualisierten und mit mid identifizierten Modells beginnt mit der Devirtualisierung dessen Graphanteils, wodurch der Graph m entsteht. Dieser wird anschließend um die Bestandteile Cnst und inherit erweitert.

Definition 5.14: Virtualisierungsmodell für Produktionsregel-Constraints
Für das Konzept der Constraints für Produktionsregeln ist die Definition des Virtualisierungsmodells für Produktionsregeln wie folgt anzupassen (Anpassung grün markiert):

ProdVirtModel° $=_{def}$ ProdVirtModel +



Definition 5.15: Devirtualisierungsfunktion für Constraints
Für das Konzept der Constraints für Produktionsregeln ist die Definition der Devirtualisierungsfunktion wie folgt anzupassen:

devirtualizeprod*(g,pid) $=_{def}$ g'[Cnst$\mapsto$C'] mit
  g' = devirtualizeprod(g,pid),
  C' = $\bigcup$ c $\in$ instances(Bedingung,g)$\cap$collectors(pid,in,g) : { (ct,fp) }
wobei
  ct = any(collect(c,bterm,g))
  fp = $\bigcup$ v $\in$ instances(BVar,g)$\cap$collectors(c,in,g) :
    {(any(collect(v,varname,g)) , h(any(collect(v,belegemit,g))))}
  h aus der Berechnung von devirtualizeprod(g,pid) zu entnehmen ist

5.2.4 Induktiv inkrementelle Transformation

Im Szenario der inkrementellen Transformation (vgl. Kapitel 2.3.1) wurde die parallele Weiterentwicklung zweier Graphen g0 und g1 betrachtet, wobei sich g1 aus g0 durch eine Transformation tr ergab. Änderungen an g0 führten zur Notwendigkeit, die Transformationsanwendung zu aktualisieren, wobei die zwischenzeitlichen Änderungen an g1 nicht verloren gehen sollten. Änderungen an g1 hatten jedoch keine weiteren Auswirkungen in diesem Szenario.

Bei der induktiv inkrementellen Transformation wird nicht zwischen g0 und g1 unterschieden. Es gibt also nicht zwei Graphen, die parallel weiterentwickelt werden, sondern nur einen Graphen g. Die manuellen Änderungen an diesem Graphen werden damit – anders als im Szenario der inkrementellen Transformation – ebenfalls Gegenstand der Transformation tr. Jede Transformationsanwendung wirkt damit wie ein Induktionsschritt – woraus die Namensgebung dieses Szenarios resultiert. Als konkretes Beispiel sei Abbildung 5.24 betrachtet.

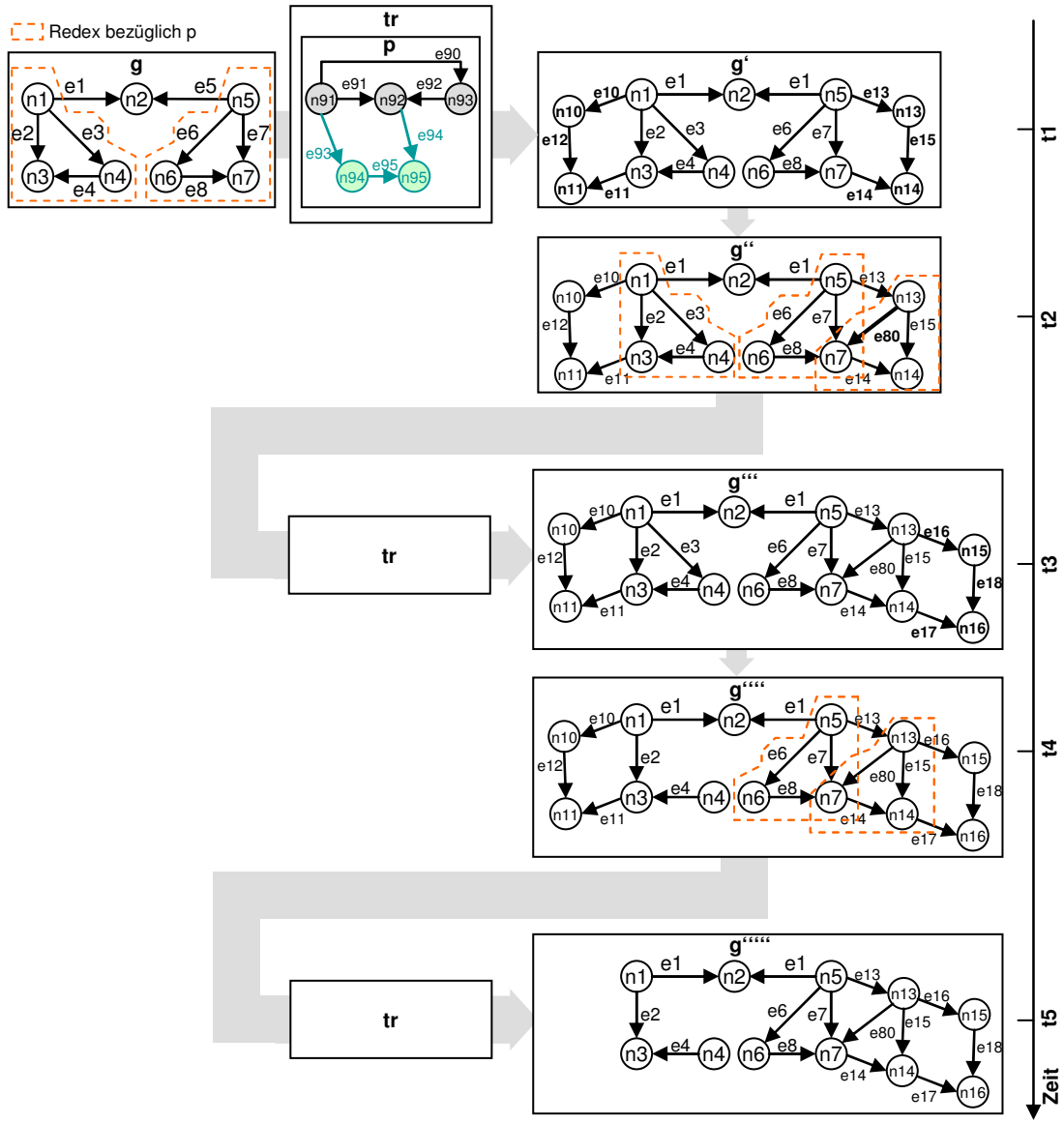


Abbildung 5.24 Motivation zur induktiv inkrementellen Transformation

Bei der ersten Transformationsanwendung (zum Zeitpunkt t1) werden im Graph g zwei Redexe gefunden und reduziert[43], wodurch der g' entsteht. Anschließend (zum Zeitpunkt t2) wird durch den Anwender die Kante e80 manuell hinzugefügt, wodurch g'' entsteht. Nun, zum Zeitpunkt t3, wird erneut Transformiert – jedoch nicht mit g, sondern mit g''. In diesem werden drei Redexe gefunden, von denen zwei dieselben sind wie bei der ersten Transformationsanwendung. Entsprechend ist nur der durch die Kante e80 neu entstandene Redex zu reduzieren, wodurch in g''' die Graphelemente e16-e18, n15 und n16 entstehen. Zum Zeitpunkt t4 wird vom Anwender eine weitere manuelle Veränderung vorgenommen, bei der die Kante e3 entfernt wird. Es entsteht g''''. In der daraufhin zum Zeitpunkt t5 folgenden Transformationsanwendung ist ein früher gefundener Redex (der e90 auf e3 abbildete), nicht mehr vorhanden. Entsprechend werden die „damals" erzeugten Elemente (e10-e12, n10 und n11) entfernt.

Der Einfachheit halber werden nur nicht-löschende Produktionsregeln betrachtet. Aus formaler und allgemeiner Sicht wird bei einer inkrementellen Transformationsanwendung die vorherige „wiederholt", wobei mit Hilfe der Transformationshistorie für dieselben Redexe dieselben Ids „erzeugt" werden. Bei der induktiv inkrementellen Transformation wird jedoch nicht mit der Eingabe, sondern mit der (modifizieten) Ausgabe gestartet (um die Induktion zu ermöglichen). In dieser Ausgabe sind die Ergebnisse der Reduktionen (also neu erzeugte Elemente) bereits enthalten. Damit erübrigt sich die Wiederholung der entsprechenden Reduktionen. Andererseits könnten Redexe verschwinden. In solchen Fällen ist die entsprechende Reduktionsanwendung rückgängig zu machen. Folglich wird mit einer induktiv inkrementellen Transformation eine frühere Transformationsanwendung nicht wiederholt, sondern nur die Ausgabe hinsichtlich des Hinzukommens oder des Verschwindens von Redexmorphismen aktualisiert.

Definition 5.16: Rückreduktion
Gegeben sei ein Graph g ∈ GRAPH*, eine Produktionsregel* p ∈ PRODRULE *und ein Regelmorphismus* hr ∈ RULEMORPH *bezüglich* g *und* p*. Die* ***Rückreduktion*** *des Graphen* g *bezüglich* hr*, notiert als* unreduce(g,p,hr)*, entfernt die durch* hr *erzeugten Elemente aus* g*, und ist wie folgt definiert:*

$$\text{unreduce}(g,p,hr) =_{def} g \restriction_G (\underline{Id}_g \,/\, (WB(hr) \,/\, WB(\text{redexmorph}(hr,p))))$$

Definition 5.17: Induktiv inkrementelle Transformationsanwendung
Gegeben sei eine Transformation tr ∈ TRANS*, ein Graph* g ∈ GRAPH *und eine Transformationshistorie* c ∈ TRANSHISTORY*. Die* ***induktiv inkrementelle Transformations-anwendung*** *bezüglich* g*,* tr *und* c*, notiert als* indinktrans(g,tr,c)*, ist wie folgt definiert:*

$$\text{indinktrans}(g,tr,c) =_{def} \text{falls } (K/K_0 \cup K_0/K) = \emptyset \text{ dann } (g,c) \text{ sonst indinktrans}(g', tr, c')$$

wobei

$$K = [\bigcup_{p \in tr}: \{p\} \times pmf(p,g)]$$

$$K_0 = (p,hr) \in c : \{ (p,\text{redexmorph}(hr,p)) \}$$

$$(p,h) = rsf(K/K_0 \cup K_0/K)$$

[falls $(p,h) \in K/K_0$ dann
  $(g',hc) = reduce(g,p,h)$
  $c' = c \cup \{(p,h \cup hc)\}$]
sonst
falls $(p,h) \in K_0/K$ dann
  $g' = unreduce(g,p,h)$
  $c' = c / \{(p,h)\}$]

[43] Es wird eine redexunikate Transformationsanwendung ausgeführt.

Mit K werden die aktuell vorhandenen, und mit K_0 die bisher vorhandenen Redexmorphismen in g erfasst. Die beiden Differenzen ergeben die hinzukommenden und verschwindenden Redexmorphismen. Gemäß einer Redexauswahlfunktion wird entscheiden, welcher von diesen jeweils als nächstes verarbeitet werden soll. Hinzukommende werden reduziert, verschwindende rückreduziert. In beiden Fällen wird die Transformationshistorie entsprechend angepasst.

Definition 5.18: Szenario der induktiv inkrementellen Transformationsanwendung
Geben sei eine Transformation tr ∈ TRANS *sowie eine referenziell nicht transparante Funktion* user : GRAPH → GRAPH *die das Verhalten eines Anwenders bei der Erstellung eines neuen bzw. Veränderung eines gegebenen Graphen beschreibt. Das* ***Szenario der induktiv inkrementellen Transformationsanwendung*** *bezüglich* tr *und* user*, notiert als* $scenindink_{user}(tr)$*, ist wie folgt definiert:*

$scenindink_{user}$(tr) $=_{def}$ scenindink(g, tr, c)
wobei
g = user(∅)
(g',c) = indinktrans(g, tr, ∅)

$scenindink_{user}$(g, tr, c) $=_{def}$ $scenindink_{user}$(g''', tr, c')
wobei
g' = user(g)
(g1'',c') = indinktrans(g', tr, c)
g''' = resolveconflicts(g'')

Das Szenario der induktiv inkrementellen Transformationsanwendung unterscheidet sich von der zur inkrementellen Transformationsanwendung durch das Zusammenführen von g0 und g1 zu einem einzigen Graphen g. Da durch die Rückreduktion auch hier Konflikte auftreten können, ist im Allgemeinen eine einer Funktion zur Konfliktauflösung zu verwenden. Dieser können – genauso wie bei der inkrementellen Transformationsanwendung – noch weitere Parameter übergeben werden, wie z.B. die gesamte Historie.

5.3 Automatisierungskonzept

Die Benennung der inkrementellen Transformation als Werkzeug genügt allein nicht, um die Teilprobleme I und II der Aufgabenstellung zu lösen. Es Bedarf einer genauen Beschreibung wie der zu transformierenden Graph und die dazu anzuwendenden Produktionsregeln im Bezug auf die Problemdomäne zu organisieren sind.

Wie es der Lösungsansatz bereits skizziert hat, wird die Produktbibliothek sowohl als Transformationsgegenstand als auch als Quelle für die anzuwendenden Produktionsregeln verwendet. Durch die Formalisierung der Vorgaben entspricht diese Konstellation nicht mehr der in der Problemstellung definierten Produktbibliotheksstruktur, sondern stellt eine Erweiterung dar. Die Struktur einer erweiterten Produktbibliothek wird im Kapitel 5.3.1 fixiert. Erst auf dieser Grundlage erfolgt die Festlegung der von der Aufgabenstellung gesuchten Funktionen steuerung (in Kapitel 5.3.2) und planung (im Kapitel 5.3.3) zur Automatisierung der zugehörigen Schritte im Rahmen des Projektcontrollings.

Für die Automatisierung bzw. Unterstützung der Kontrolle sind ebenfalls Erweiterungen des Produktbibliotheksmodells notwendig. Dies wird ebenfalls in Kapitel 5.3.1 ausgeführt. Auf dieser Grundlage erfolgt die Festlegung der von der Aufgabenstellung gesuchten Funktionen kontrolle_V und kontrolle im Kapitel 5.3.4.

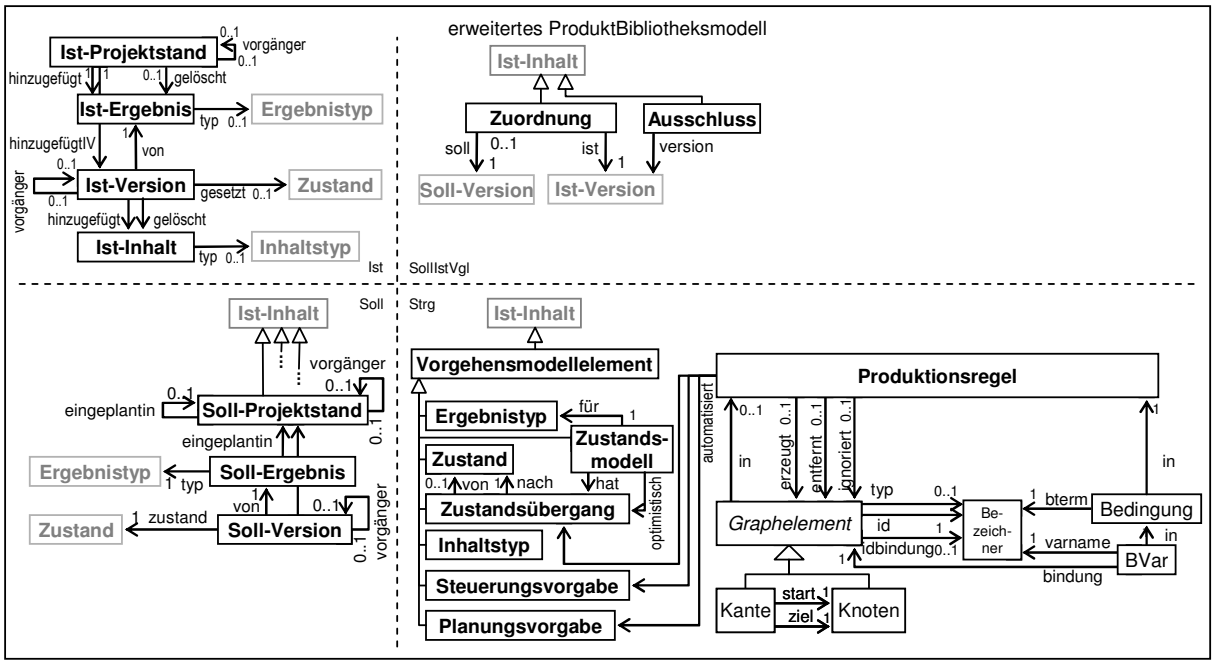
5.3.1 Erweiterte Produktbibliothek

Die erweiterte Produktbibliothek verwendet sämtliche Konzepte der Grundlagen und der Grundlagenerweiterungen aus Kapitel 5.2. Das Produktbibliotheksmodell wird um das Virtualisierungsmodell erweitert, welches durch Assoziationen zwischen dem Konstrukt Produktionsregel und den Vorgabenkonstrukten (einschließlich der Zustandsübergänge) verbunden wird. Im Teil Strg kommt weiterhin eine Assoziation hinzu, um Zustandsübergänge als optimistisch markieren zu können. Im Teil SollIstVgl kommt die Entität Ausschluss hinzu, um Ist-Versionen von der Erzeugung von Zuordnungsvorschlägen ausschließen zu können. Auf Ebene der Instanzen erhält jede Produktbibliothek die Transformationshistorie als Bestandteil, so dass dessen Aufenthaltsort geregelt ist.

Da die Produktbibliothek eine Historie über alle Ist-Projektstände führt, werden durch die inkrementellen Transformation vorgenommene Änderungen nicht direkt, sondern virtualisiert (d.h. als virtuelle Hinzufügungen und Löschungen) in die neue Produktbibliothek überführt. Dazu wird in diesem Kapitel die Änderungsvirtualisierung als Technik definiert, die bei der Automatisierung der Steuerung, Planung und Kontrolle Anwendung findet.

Definition 5.19: Produktbibliotheksmodell
Für die Lösung ist die Definition des Produktbibliotheksmodells wie folgt anzupassen:

$$\text{Bibomodell}^{\circ} =_{def}$$



Definition 5.20: Produktbibliothek mit Automatisierung
Für die Lösung ist die Definition der Menge BIBO *wie folgt anzupassen:*

$$BIBO^{\circ} = \{ B \in BIBO \mid \underline{hist}_B \in TRANSHISTORY \}$$

Produktbibliotheken werden um den Bestandteil hist, eine Transformationshistorie, erweitert.

Definition 5.21: Arbeiten mit Produktbibliothek in OCL
Zum einfachen Formulieren von Produktionsregeln seien für die algebraischen Definitionen aus 4.2.1 in Form der in Abbildung 5.25 definierten OCL-Funktionen vereinbart.

```
def istvorgIV(iv,ivv:Ist-Version): Boolean = iv.vorgänger->contains(ivv)

def istvorgSV(sv,svv:Soll-Version): Boolean = sv.vorgänger->contains(svv) or
  sv.vorgänger->isEmpty() and
    Soll-Version.allInstances()->collect( v | v.vorgänger=svv )->isEmpty()
      and istvorgSP(svv.eingeplantin,sv.eingeplantin)
def: istvorgSP(sp,spv:Soll-Projektstand): Boolean = sp.vorgänger->contains(spv) or
  sp.vorgänger->isEmpty() and
    Soll-Projektstand.allInstances()->collect( sp1 | sp1.vorgänger=spv )->isEmpty()
      and istVorgSP(sp.eingeplantin,spv.eingeplantin)

def vorgIV(iv:Ist-Version): Set(Ist-Version) = Ist-Version.allInstances()->collect( ivv | istvorg(iv,ivv) )

def vorgSV(sv:Soll-Version): Set(Soll-Version) = Soll-Version.allInstances()->collect( svv | istvorg(sv,svv) )

def vorgPrjstd(ip:Ist-Projektstand): Set(Ist-Projektstand) = ip.vorgänger->asSet()

def inhalteIV(iv:Ist-Version): Set(Ist-Inhalt) = if vorgIV(iv)->isEmpty() then Set{} else
  inhalteIV( vorgIV(iv)->any() )->union(ip.hinzugefügt->asSet()) - ip.gelöscht->asSet() endif

def inhaltePrjstdIV(ip:Ist-Projektstand): Set(Ist-Version) = if vorgPrjstd(ip)->isEmpty() then Set{} else
  inhaltePrjstdIV( vorgPrjstd(ip)->any() )->union(ip.hinzugefügtIV->asSet()) –
    Ist-Version.allInstances()->collect( iv | ip.gelöscht->contains(iv.von) ) endif

def inhaltePrjstdIE(ip:Ist-Projektstand): Set(Ist-Ergebnis) = if vorgPrjstd(ip)=Set{} then Set{} else
  inhaltePrjstdIE( vorgPrjstd(ip)->any() )->union(ip.hinzugefügt->asSet()) – ip.gelöscht->asSet() endif

def zustandIV(iv:Ist-Version): Zustand = if ! iv.gesetzt.oclIsUndefined() then iv.gesetzt else zustandIV(iv.vorgänger) endif

def aktversionIE(ie:Ist-Ergebnis): Set(Ist-Version) = Ist-Version->any( iv | iv.von=ie and ! Ist-Version.allInstances()->exists( ivn |
      transvorgIV(ivn)->contains(iv) and ivn.von=ie ) )

aktPrjstd(): Ist-Projektstand = Ist-Projektstand->any( ip | ! Ist-Projektstand.allInstances()->exists( ipn | ipn.vorgänger=ip) )
```

Abbildung 5.25 OCL-Funktionen für Produktionsregeln für Produktbibliothek (1/2)

Zur Kurznotation für den Zugriff auf den aktuellsten Projektstand seien die in Abbildung 5.26 definierten Constraints vereinbart.

```
def aktinhalte(): Set(Ist-Inhalt) = Ist-Inhalt.allInstances()->collect( ii | inhaltePrjstdIV(aktPrjstd())->exists( iv | inhalteIV(iv)->contains(ii) ) )

def aktergebnisse(): Set(Ist-Ergebnis) = inhaltePrjstdIE(aktPrjstd())
```

Abbildung 5.26 OCL-Funktionen für Produktionsregeln für Produktbibliothek (2/2)

Im Weiteren wird implizit davon ausgegangen, dass die definierten OCL-Funktionen in jeder Produktionsregel enthalten sind.

Definition 5.22: Produktionsregeln aus Produktbibliothek
Sei eine Produktbibliothek B ∈ BIBO *und ein* v ∈ instances(y,B) *mit* y ∈ {Planungsvorgabe, Steuerungsvorgabe, Zustandsübergang} *gegeben. Die* ***Produktionsregeln*** *für* v ***aus*** B*, notiert mit* prodrules(B,v)*, sind wie folgt definiert:*

$$\text{prodregeln}(B,v) =_{def} \bigcup i \in \text{instances}(\text{Produktionsregel},B) \text{ mit } v \in \text{collect}(i,\text{automatisiert},B) : \text{devirtualizeprod}(B,i)$$

Bei der Devirtualisierung werden nur solche Produktionsregeln in B betrachtet, die zu einem v zugeordnet sind.

Definition 5.23: Impliziter Bezug auf Enthaltensein im aktuellsten Ist-Projektstand
Sei eine Produktionsregel p ∈ PRODRULE *gegeben. Deren Fassung mit explizitem* ***Bezug*** *auf das Enthaltensein von Elementen im aktuellsten Projektstand wird mit* explakt(p) *notiert und ist wie folgt definiert:*

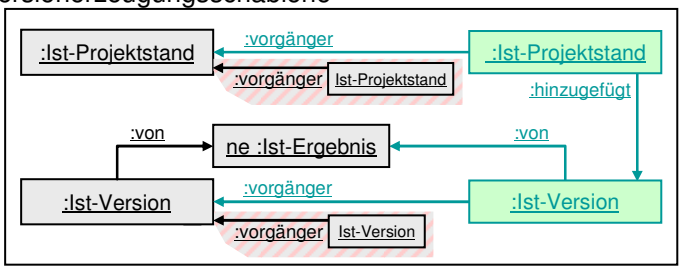
$$\text{explakt}(p) =_{def} p' = p[\underline{\text{Cnst}} \mapsto C1 \cup C2]$$
wobei
$$C1 = \bigcup y \in Y1: [\bigcup j \in \text{instances}(y,\text{searchpattern}(p)): \{(ct1,fp1)\}]$$
$$ct1 = \text{'aktinhalte()->contains(x)'}$$
$$fp1 = \{ \text{'x'} \mapsto j \}$$
$$Y1 = \underline{y}_{Strg} \cup \underline{y}_{Soll} \cup \underline{y}_{SollIstVgl}$$

$$C2 = \bigcup y \in Y2: [\bigcup j \in \text{instances}(y,\text{searchpattern}(p)): \{(ct2,fp2)\}]$$
$$ct2 = \text{'aktinhalte()->contains(x)'}$$
$$fp2 = \{ \text{'x'} \mapsto j \}$$
$$Y2 = \{\text{Ist-Ergebnis}\}$$

Die Fassung p' einer Produktionsregel p unterscheidet sich durch explizit hinzugefügte Constraints, deren Vorhandensein in p daher nur als implizit angenommen wurde. Ein solcher Constraint besagt, dass die Belegung eines Suchmusterknotens j im aktuellen Ist-Projektstand vorhanden (und auch nicht virtuell wieder gelöscht) ist. Für welche Suchmusterknoten ein solcher Constraint in der expliziten Fassung jeweils hinzugefügt wird, ist durch deren Typ und die Menge Y1 bzw. Y2 bestimmt. Konkret sind dies hier alle Elemente der Teilbibliotheken Strg, Soll und SollIstVgl, sowie Ist-Ergebnisse aus der Teilbibliothek Ist. Ist-Inhalte werden dagegen gerade nicht implizit auf das Enthaltensein im aktuellsten Ist-Projektstand beschränkt, da für diese Produktionsregeln auch nach ihrem (virtuellen) Löschen noch greifen können sollten – wie im Lösungsansatz mit Bezug auf unterschiedliche Projektphasen gezeigt.

Definition 5.24: Änderungsvirtualisierung
Seien zwei Produktbibliothek B, B' ∈ BIBO *gegeben. Die* ***Virtualisierung der Änderungen*** *von* B' *gegenüber* B *bezüglich einer Teilbibliothek* tid ∈ {BiboSoll, BiboStrg, BiboSollIstVgl} *wird mit* ändvirt(B,B',tid) *notiert, und ist wie folgt definiert:*

ändvirt(B,B',tid) $=_{def}$ B'' + L mit
B'' = π_1(reduce(B',{p},any(pmf(p,B')))), einmalig ausgeführt
p = Istversionerzeugungsschablone[idm↦{ne↦tid}]
L = { Id↦Id', s↦s', t↦t', y↦y' } mit
Id' = $Id^+ \cup WB(edgeadd) \cup WB(edgedel)$
s' = $[\bigcup_{i \in Id^+} : \{edgeadd(i) \mapsto v\}] \cup [\bigcup_{i \in Id^-} : \{edgedel(i) \mapsto v\}] \cup (s_{B'}/s_B)$
t' = $[\bigcup_{i \in Id^+} : \{edgeadd(i) \mapsto i\}] \cup [\bigcup_{i \in Id^-} : \{edgedel(i) \mapsto i\}] \cup (t_{B'}/t_B)$
y' = $[\bigcup_{i \in Id^+} : \{edgeadd(i) \mapsto$ hinzugefügt$\}] \cup [\bigcup_{i \in Id^-} : \{edgedel(i) \mapsto$ gelöscht$\}] \cup (y_{B'}/y_B)$
wobei
v = latestversion(tid,B'')
Id^+ = instances(Ist-Inhalt,B') / instances(Ist-Inhalt,B)
Id^- = instances(Ist-Inhalt,B') / instances(Ist-Inhalt,B')
edgeadd(i) = $\bigcup_{i \in Id^+}$: {(i,newGE())}, einmalig ausgeführt
edgedel(i) = $\bigcup_{i \in iId^-}$: {(i,newGE())}, einmalig ausgeführt
Istversionerzeugungsschablone =



Diese Funktion baut auf der Situation auf, dass B' aus B hervorgegangen ist, wobei nur in der Teilbibliothek tid Änderungen von Ist-Inhalten vorgenommen wurden. Diese werden durch die beiden Mengen Id^+ (für Hinzufügungen) und Id^- (für Löschungen) erfasst. Anschließend werden diese Änderungen durch eine neue Version für tid, sowie der Nutzung von Kanten mit den Typen hinzugefügt bzw. gelöscht, virtualisiert. Zu beachten ist, dass von Ist-Inhalten nur der jeweilige Knoten (vom Typ Ist-Inhalt) als hinzugefügt oder gelöscht markiert wird. An Ist-Inhalten hängende Kanten werden bei neuen Ist-Inhalten implizit in das neue B'' übernommen. Bei virtuell gelöschten Ist-Inhalt-Knoten werden die daran hängenden Kanten ebenfalls implizit als gelöscht verstanden.

5.3.2 Automatisierte Planung

Die Automatisierung der Planung besteht darin, die in der gegebenen Produktbibliothek virtualisiert vorliegenden Produktionsregeln (die zu Planungsvorgaben und Zustandsübergangsbedingungen zugeordnet sind) zunächst zu devirtualisieren und dann auf die Produktbibliothek anzuwenden. Der dabei zu verwendende Transformationshistorie wird aus der Produktbibliothek entnommen, und nach der Transformation durch die neue Transformationshistorie ersetzt.

Zunächst sei die Automatisierung des Zustandsmodells im Detail erläutert. Abbildung 5.27 zeigt dazu ein Beispiel mit vier Produktionsregeln die im Sinne eines Prädikates zu jeweils einem Zustandsübergang eine Zustandsübergangsbedingung bilden.

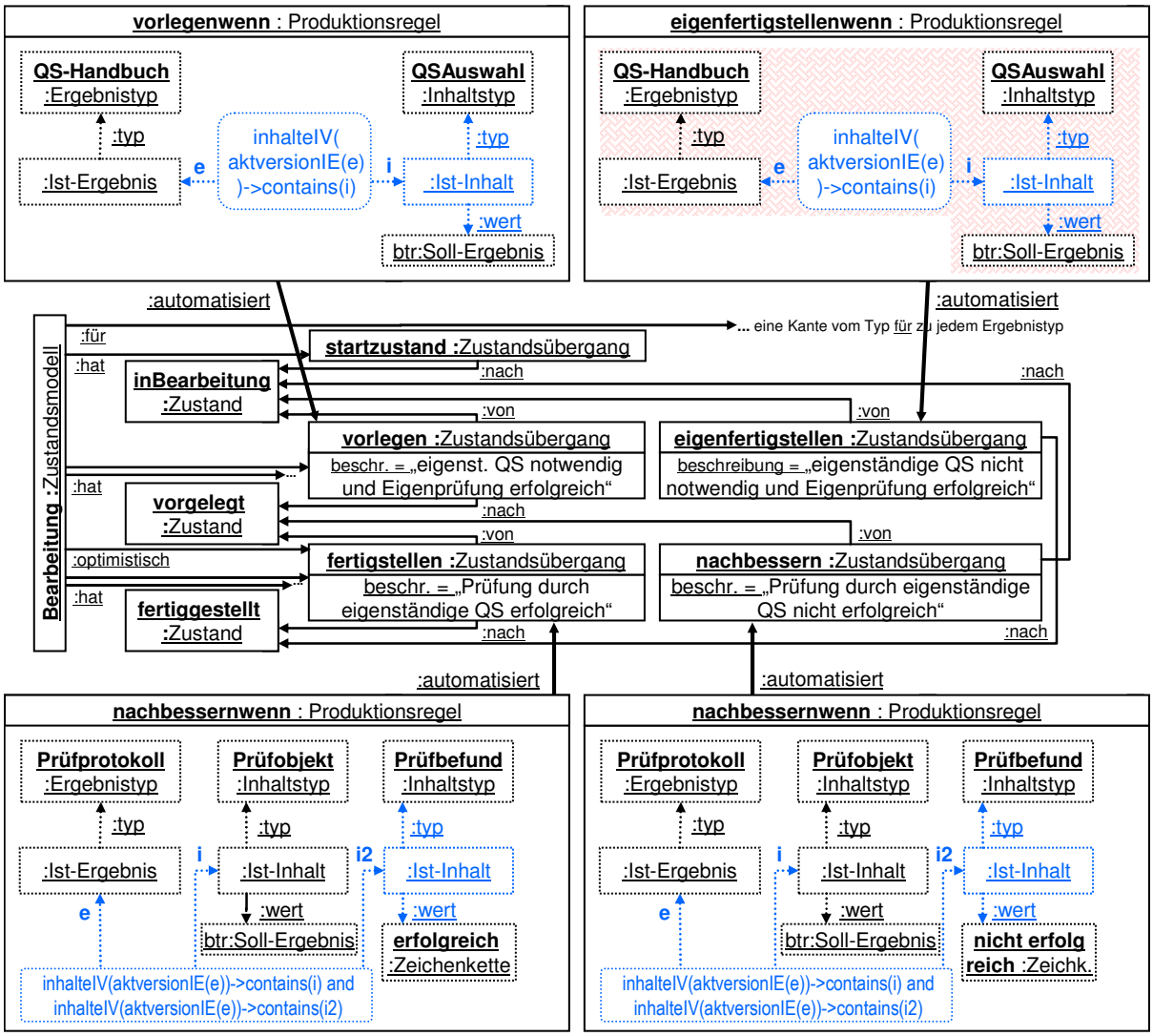


Abbildung 5.27 Beispiel eines Zustandsmodells mit automatisierenden Produktionsregeln

Zur Ableitung der bei der inkrementellen Transformation effektiv eingesetzten Produktionsregeln werden die in Abbildung 5.28 gezeigten Schablonen verwendet. Die in der Abbildung linke Produktionsregel dient zur Erzeugung einer Soll-Version für den Startzustand, die rechte für jeden Folgezustand. Die effektiven Produktionsregeln entstehen durch das Einsetzen der Zustandsübergangsbedingungen in diese Schablonen. Abbildung 5.29 zeigt dies am Beispiel der Zustandsübergangsbedingung vorlegenwenn.

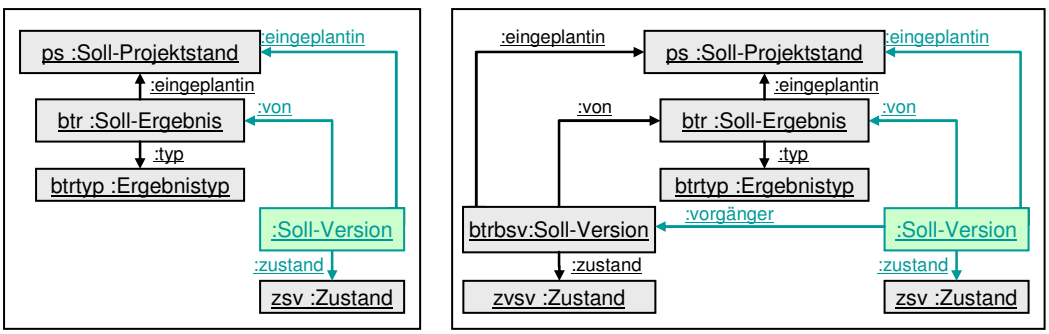


Abbildung 5.28 Schablonen für Produktionsregeln zur Einplanung von Soll-Versionen

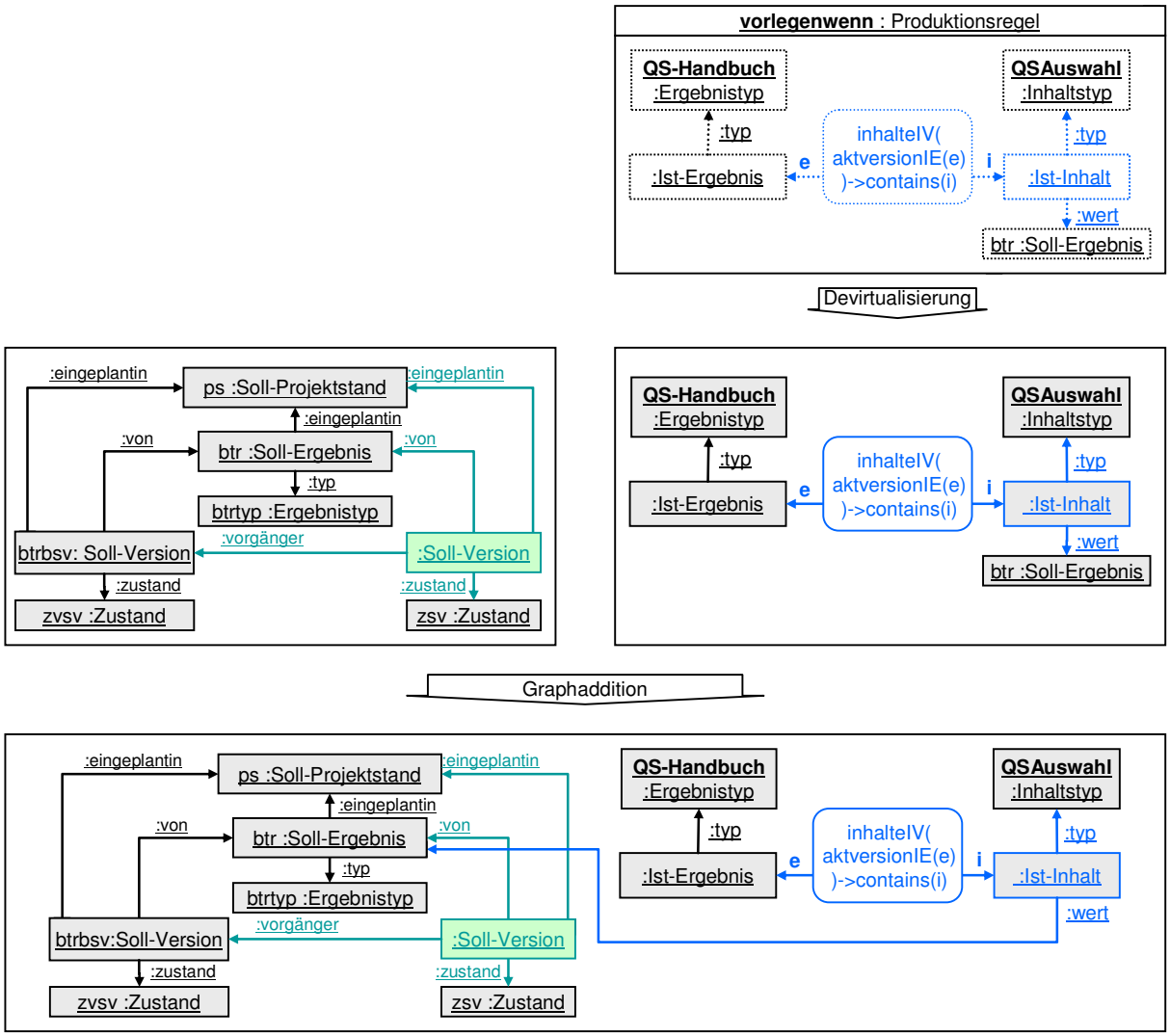


Abbildung 5.29 Effektive Produktionsregeln für Erzeugung von Soll-Versionen

Die Kopplung zwischen den Schablonen und den Zustandsübergangsbedingungen erfolgt durch Ids, wie z.B. btr (für „*Betr*achtetes Soll-Ergebnis“), bsvbtr (für „*b*isherige *S*oll-*V*ersion des *betr*achteten Soll-Ergebnisses). Ein in einer Zustandsübergangsbedingung enthaltener Suchmusterknoten mit der Id btr fällt beim Einsetzen in die Schablone mit dem Knoten zusammen, der die gleiche Id hat. Analog verhält sich dies auch mit bsvbtr sowie den anderen Ids der Schablone.

Die Eigenschaft der optimistischen Einplanung wird allerdings erst dadurch erreicht, dass für als optimistisch markierte Zustandsübergänge gerade nicht die ursprüngliche Zustandsübergangsbedingung verwendet wird, sondern die Unerfülltheit aller dazu alternativen Zustandsübergänge. Damit kann zunächst die entsprechende Soll-Version eingeplant werden – noch lange bevor eines der Übergänge tatsächlich durchgeführt werden kann. Als Beispiel sei dazu Abbildung 5.30 betrachtet: In der Produktbibliothek B10 existiert die Einplanung eines Soll-Ergebnisses vom Typ Systemarchitektur. In der darauffolgenden inkrementellen Transformation wird die Produktbibliothek B11 erzeugt, in der sämtliche zu erwartenden Soll-Versionen optimistisch eingeplant werden – konkret sind dies die Versionen sb50 für den Zustand inBearbeitung und die Version sv51 für die den Zustand fertiggestellt – gemäß dem Zustandsmodells aus Abbildung 5.27.

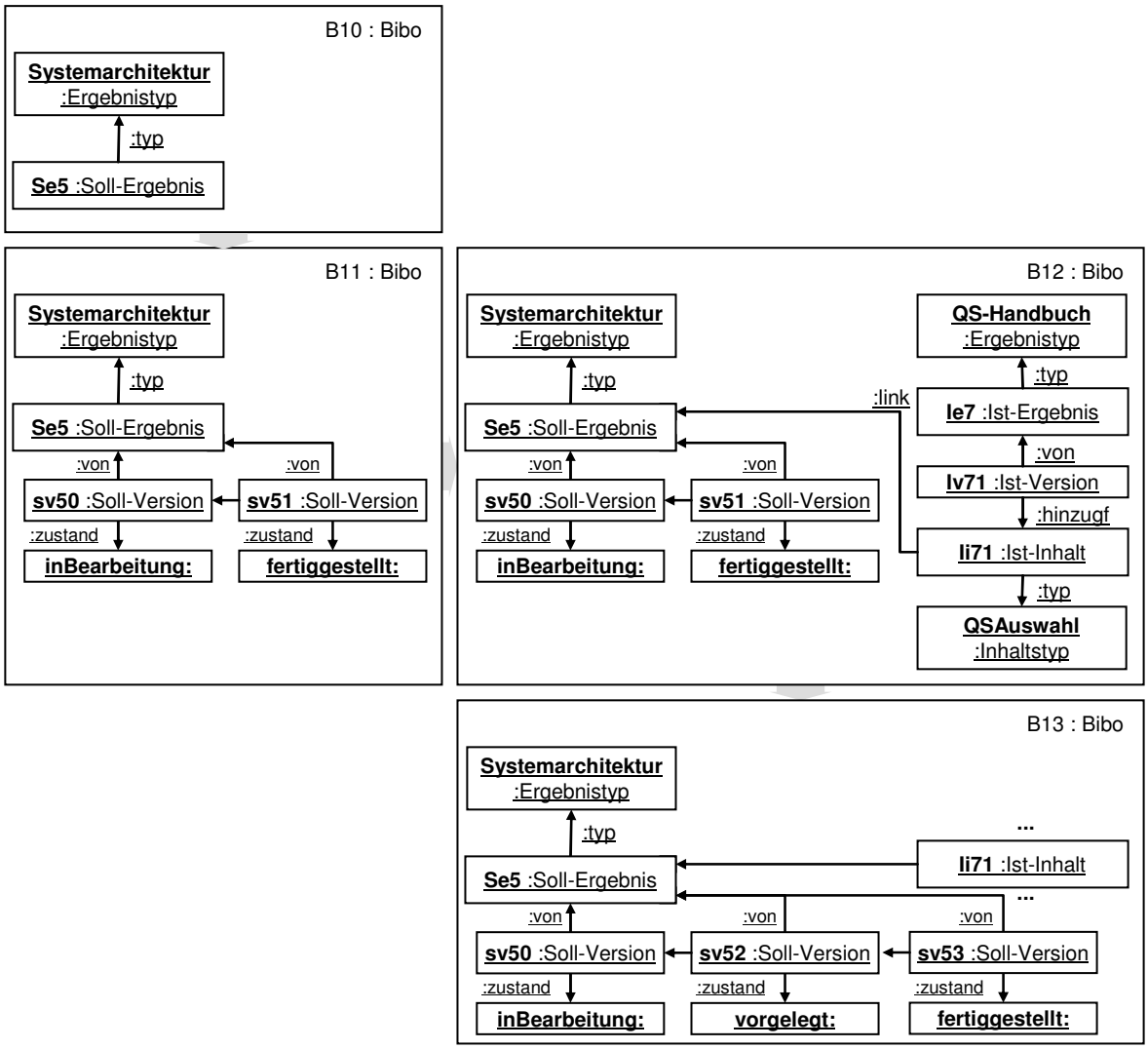


Abbildung 5.30 Effektive Wirkung der optimistischen Planung am Beispiel

Zu einem späteren Zeitpunkt, mit der Produktbibliothek B12, ist ein QS-Handbuch hinzugekommen, was genau das gewählte Soll-Ergebnis referenziert – wodurch der Optimismus, auf eine eigenständige Qualitätssicherung zu verzichten, sich hier nicht als gerechtfertig herausgestellt hat. Bei der darauffolgenden inkrementellen Transformation wird nun, mit der Produktbibliothek B13, die Soll-Version-Einplanung sv51 automatisch zurückgebaut, und durch eine den neuen Bedingungen angepassten optimistisch eingeplanten Soll-Versionen ersetzt.

Da auch die Änderungen des Solls versioniert werden, stellt die Abbildung 5.30 den aktuellsten Stand nur effektiv dar. Der tatsächliche Inhalt der Produktbibliothek ist in Abbildung 5.31 dargestellt.

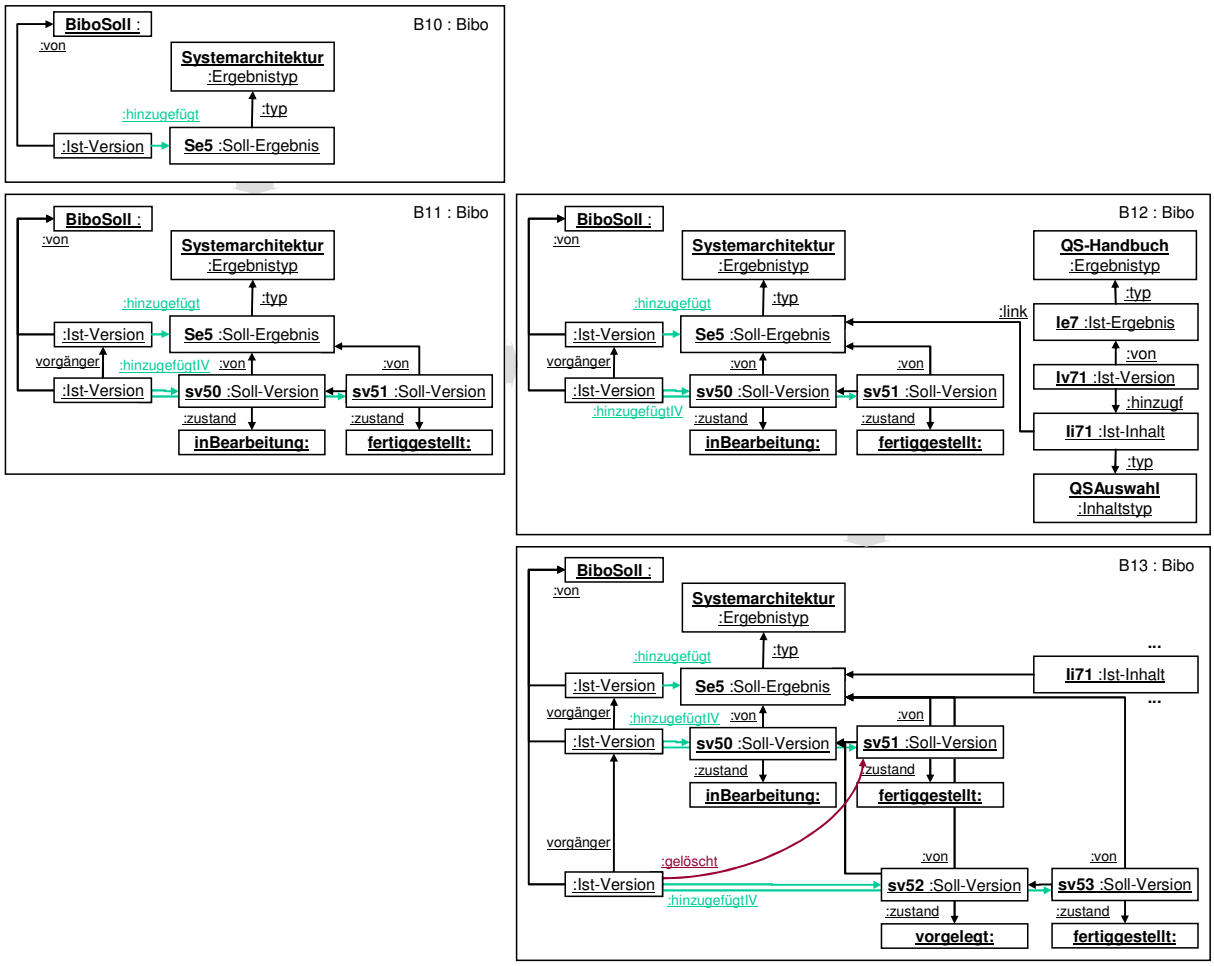


Abbildung 5.31 Tatsächliche Wirkung der optimistischen Planung am Beispiel

Definition 5.25: Schablone für Versionsplanung

Gegeben seien eine Produktbibliothek B ∈ BIBO *und ein Zustandsübergang* zü ∈ instances(Zustandsübergang,B). *Die eingesetzte* ***Schablone*** *für* zü *bezüglich* B*, notiert mit* schablone(zü,B)*, ist wie folgt definiert:*

$$
\begin{aligned}
&\text{schablone}(zü,B) =_{def}\\
&\quad \text{falls } \text{collect}(zü,\text{von},B)=\emptyset \text{ dann}\\
&\qquad \text{template}_0[\underline{\text{idm}} \mapsto \{\text{zsv} \mapsto \text{any}(\text{collect}(zü,\text{nach},B)\}]\\
&\quad \text{sonst}\\
&\qquad \text{template}_F[\underline{\text{idm}} \mapsto \{\text{zsv} \mapsto \text{any}(\text{collect}(zü,\text{nach},B),\ \text{zvsv} \mapsto \text{any}(\text{collect}(zü,\text{von},B)\}]\\
&\text{wobei}\\
&\quad \text{schablone}_0 = \text{Produktionsregel aus Abbildung 5.28, links}\\
&\quad \text{schablone}_F = \text{Produktionsregel aus Abbildung 5.28, rechts}
\end{aligned}
$$

Mit der Funktion template wird die passende Schablone anhand des Vorhandenseins eines Vorgängerzustands ausgewählt. Zur einfacheren Handhabung wird der neue Zustand (sowie der vorherige, falls zutreffend) in die Schablone eingesetzt. Unaufgelöst bleibt jedoch noch der Ergebnistyp, was erst an anderer Stelle an eine konkrete Id gebunden wird.

Definition 5.26: Effektive Produktionsregel für realistische Versionsplanung

Gegeben seien eine Produktbibliothek B ∈ BIBO *und ein Zustandsübergang* zü ∈ instances(Zustandsübergang,B). *Die* ***Produktionsregel für die realistische Versionsplanung*** *zu* zü *bezüglich* B*, notiert mit* effprodreal(zü,B)*, ist wie folgt definiert:*

$$\text{effprodreal}(zü, B) =_{def} \text{template}(zü,B) + [\sum r \in \text{prodrules}(B,zü) : r]$$

Die effektive Produktionsregel für die realistische Versionsplanung ergibt sich durch Anwendung der entsprechenden Schablone, zu der zum Zustandsübergang zugeordnete(n) Zustandsübergangsbedingung(en) hinzuaddiert werden (unter Verwendung der Graphdefinion nach Definition 2.17).

Definition 5.27: Effektive Produktionsregel für optimistische Versionsplanung
Gegeben seien eine Produktbibliothek B ∈ BIBO*, ein Zustandsmodell* zm ∈ instances(Zustandsmodell,B)*, und ein Zustandsübergang* zü *zu* zm*, d.h. mit* zü ∈ collect(zm,hat,B). *Die* ***Produktionsregel für die optimistische Versionsplanung*** *zu* zü *bezüglich* zm *und* B, *notiert mit* effprodopt(zü,zm,B)*, ist wie folgt definiert:*

$$\text{effprodopt}(zü,zm,B) =_{def} \text{template}(zü,B) + [\sum r \in [\bigcup azü \in A : \text{prodrules}(B,azü)] : \text{negate}(r)]$$
wobei
$A = \{ t \in \text{collect}(zm,\text{hat},B) \mid \text{collect}(t,\text{von},B) = \text{collect}(zü,\text{von},B) \} / \{zü\}$
$\text{negate}(r) = r [\underline{\text{neg}} \mapsto (\underline{\text{Id}}_r / \underline{\text{Id}}_{\text{template}(zü,B)}) \cup \{\underline{\text{neg}}_r\}]$

Die effektive Produktionsregel für die optimistische Versionsplanung wird prinzipiell wie die zur realistischen Versionsplanung ermittelt. Der Unterschied besteht darin, dass nicht die zu zü zugeordneten Zustandsübergangsbedingungen zur Schablone addiert werden, sondern die dazu alternativen (A) in negierter Form. Welche Zustandsübergänge die zu zü alternativen sind, kann je nach Zustandsmodell anders ausfallen. Daher wird das zu betrachtende Zustandsmodell hier als (gegenüber effprodreal zusätzlicher) Parameter entgegengenommen. Die Negation einer Zustandsübergangsbedingung r erfolgt durch die Konstruktion eines neuen Negationsbereiches der die gesamte Regel r umfasst (mit Ausnahme Elementen die auch in der Schablone vorhanden sind). Dies schließt nicht nur die Graphelemente von r, sondern auch alle dessen bisherige Negationsbereiche mit ein.

Definition 5.28: Effektive Produktionsregeln für Zustandsmodell
Gegeben sei eine Produktbibliothek B ∈ BIBO *und ein Zustandsmodell* zm ∈ instances(Zustandsmodell,B). *Die Menge der effektiven Produktionsregeln zu* zm *in* B, *notiert mit* effprods(zm,B), *ist wie folgt definiert:*

$$\text{effprods}(zm,B) =_{def} [\bigcup zü \in \text{collect}(zm,\text{hat},B) : \text{effprod}(zü,zm,B)]$$

Die ***Produktionsregel*** *für die Versionsplanung zu* zü ***bezüglich*** zm *und* B, *notiert mit* effprod (zü,zm,B)*, ist wie folgt definiert:*

$$\text{effprod}(zü,zm,B) =_{def} \text{falls } (A=\emptyset) \text{ oder } (A \cup \{zü\}) \cap \text{collect}(zm,\text{optimistisch},B) = \{zü\}$$
dann effprodopt(zü,zm,B)
sonst effprodreal(zü,B)
wobei
$A = \{ t \in \text{collect}(zm,\text{hat},B) \mid \text{collect}(t,\text{von},B) = \text{collect}(zü,\text{von},B) \} / \{ zü \}$

Für jeden Zustandsübergang in zm wird genau eine effektive Produktionsregel erzeugt. Dazu wird zunächst entschieden, ob es eine optimistische oder realistische wird, und dann die entsprechende Konstruktion vorgenommen. Die Entscheidung basiert auf der Betrachtung der alternativen zu zü (A). Sofern es keine alternativen gibt, oder zü die einzig optimistische ist, wird die optimistische Versionsplanung verwendet. Andernfalls wird die realistische Versionsplanung verwendet.

Definition 5.29: Effektive Produktionsregeln zum Ergebnistyp und Vorgehensmodell
Gegeben sei eine Produktbibliothek B ∈ BIBO *und ein Ergebnistyp* ey ∈ instances(Ergebnistyp,B). *Die Menge der effektiven* ***Produktionsregeln zum Ergebnistyp*** ey *bezüglich* B, *notiert mit* effprods(ey,B), *ist wie folgt definiert:*

$$effprods(ey,B) =_{def} \bigcup zm \in collectors(ey,für,B) : [\bigcup p \in effprods(B,zm) : \{ p[\underline{idm} \mapsto \underline{idm}_{p(\cup)}\{btrtyp \mapsto ey\}]\}]$$

Die Menge der effektiven ***Produktionsregeln zum Vorgehensmodell*** *bezüglich* B, *notiert mit* effprods(B), *ist wie folgt definiert:*

$$effprods(B) = \bigcup ey \in instances(Ergebnistyp,B) : effprods\ (ey,B)$$

Für einen Ergebnistyp ergibt sich die Menge der effektiven Produktionsregeln durch die effektiven Produktionsregeln des zugehörigen Zustandsmodells. Da letztere noch nicht an einen Ergebnistyp gebunden sind, wird dies hier für jede dieser Produktionsregeln nachgeholt.

Definition 5.30: Automatisierte Planung
Die Funktion planung *verkörpert die* ***Automatisierung*** *von manuellen Tätigkeiten* ***des Planungsschritts*** *(im Rahmen der Aufgabenstellung) und ist wie folgt definiert:*

$$planung(B) =_{def} B''[\underline{hist} \mapsto c']$$
wobei
$$B'' = ändvirt(B,B',BiboSoll)$$
$$(B',c') = indinktrans(\ B,\ V_{(\cup)}effprods(B),\ \underline{hist}_B\ )$$
$$V = \bigcup v \in instances(Planungsvorgabe) : [\bigcup p \in prodregeln(B,v) : \{explakt(p)\}]$$

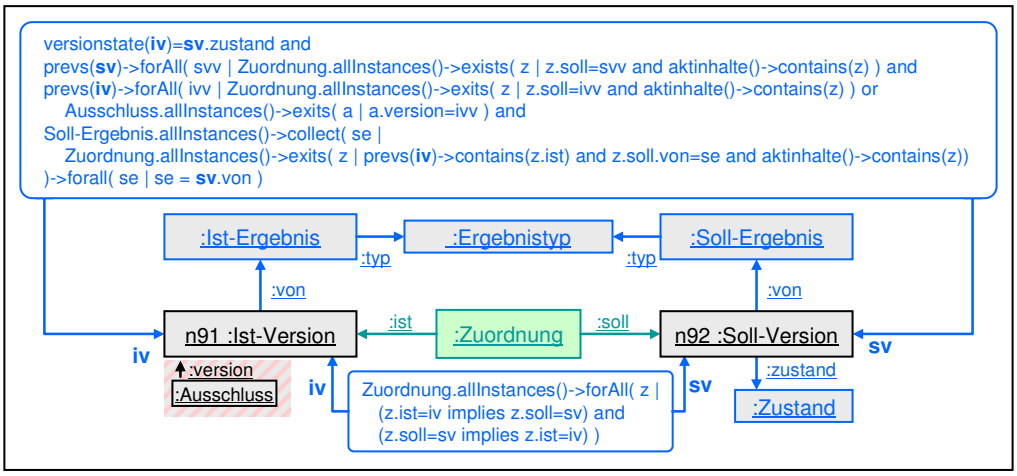
Zunächst werden die für die inkrementelle Transformation einzusetzenden Produktionsregeln bestimmt. Diese ergeben sich aus der Devirtualisierung der in B virtualisiert vorliegenden Produktionsregeln für Planungsvorgaben, sowie aus der Berechnung der effektiven Produktionsregeln für die Versionsplanungen. Der zu verwendende Transformationshistorie wird ebenfalls aus B entnommen. Das inkrementell berechnete Transformationsergebnis, die neue Produktbibliothek B' und die neue Transformationshistorie c' wird wie folgt aufbereitet: Änderungen an der Teilbibliothek Soll werden virtualisiert, wodurch B'' entsteht. Die bisherige Transformationshistorie wird durch die neuen ersetzt (die inkrementelle Transformation ändert nur die Graphanteile von B, so dass in $\underline{hist}_{B''}=\underline{hist}_B$ gilt).

5.3.3 Teilautomatisierung der Kontrolle

Die (Teil-)Automatisierung der Kontrolle basiert auf der inkrementellen Transformation mit Vorschlägen. Um dieses Konzept für die Kontrolle anzuwenden, wird eine universelle Produktionsregel zur Erzeugung von Zuordnungsvorschlägen definiert, und die vorgenommenen Änderungen der inkrementellen Transformation – analog zur Automatisierung der Planung – durch eine neue Version für die Teilbibliothek SollIstVgl virtualisiert.

Definition 5.31: Produktionregel für Zuordnungsvorschläge
Die Produktionsregel für Zuordnungsvorschläge wird als Zuordnungsregel *bezeichnet und ist wie folgt definiert:*

$$Zuordnungsregel =_{def}$$



Eine für eine Soll-Version sv in Frage kommende Ist-Version iv muss im aktuellsten Ist-Projektstand vorkommen, nicht zugeordnet, und nicht von der Ermittlung von Zuordnungsvorschlägen ausgeschlossen sein. Weiterhin muss jeder Vorgänger einer solchen Ist-Version selbst bereits zugeordnet (oder von der Ermittlung von Zuordnungsvorschlägen ausgeschlossen) sein. Eine in Frage kommende Ist-Version iv muss zudem im gleichen Zustand wie sv sein, und zu einem Ist-Ergebnis mit dem gleichen Ergebnistyp gehören, wie der Ergebnistyp des Soll-Ergebnisses, zu dem sv gehört. Schließlich müssen die Vorgänger von iv, falls vorhanden und zugeordnet, zum selben Soll-Ergebnis zugeordnet sein, zu dem sv gehört.

Für die Zuordnungsvorschläge werden nur solche Soll-Versionen betrachtet, die in dem aktuellsten Ist-Projektstand vorhanden sind, nicht zugeordnet sind, und keine unzugeordneten Vorgängerversionen haben. Damit eine einmal getroffene Zuordnung sich (bei der nächsten inkrementellen Transformationsanwendung) nicht selbst ausschließt, ist der Constraint von der Idee her wie folgt definiert: Für jeden betrachteten Redexmorphismus hx – der n91 auf eine Ist-Version iv, und n92 auf eine Soll-Version sv belegt – darf es noch keine Zuordnung für iv geben, es sei denn diese Zuordnung ordnet sv zu iv zu. Analoges gilt für sv: Es darf noch keine Zuordnung für sv geben, es sei denn diese Zuordnung ordnet iv zu sv zu. Formal wird dies in OCL durch eine Allquantifizierung mit Implikationen ausgedrückt.

Definition 5.32: Teilautomatisierte Kontrolle
Die Funktion kontrolle *verkörpert die* ***Teilautomatisierung*** *von manuellen Tätigkeiten* ***des Kontrollschritts*** *(im Rahmen der Aufgabenstellung) und ist wie folgt definiert:*

kontrolle(B) $=_{def}$ B''[hist↦c']
wobei
B'' = ändvirt(B,B',BiboSollIstVgl)
(B',c') = indinktrans°(B, {explakt(Zuordnungsregel)}, $hist_B$)
indinktrans° = indinktrans mit Verwendung einer Reduktionsstrategie rsf° statt rsf
rsf° = repräsentiert den Anwender

In der Aufgabenstellung (Abbildung 4.8) und im Lösungsüberblick (Abbildung 5.1) ist die Kontrolle mit zwei einzelnen Funktionen dargestellt. Jene Funktionen werden in der Definition der inkrementellen Transformation mit Vorgaben zusammengefasst. Die erste Funktion, zur Erzeugung der Vorschläge, entspricht der Funktion pmf (im Rahmen der inkrementellen Transformation). Die zweite, zur Anwendung der Vorschläge, entspricht der Reduktion.

5.3.4 Automatisierte Steuerung

Die Automatisierung der Steuerung ist zur Automatisierung der Planung analog, wobei anstelle der Planungsvorgaben die Steuerungsvorgaben angewendet werden.

Definition 5.33: Automatisierte Steuerung
Die Funktion steuerung *verkörpert die* ***Automatisierung*** *von manuellen Tätigkeiten* ***des Steuerungsschritts*** *(im Rahmen der Aufgabenstellung) und ist wie folgt definiert:*

$$\text{steuerung}(B) =_{def} B''[\underline{\text{hist}} \mapsto c']$$
wobei
$$B'' = \text{ändvirt}(B, B', \text{BiboStrg})$$
$$(B', c') = \text{indinktrans}(B, V, \underline{\text{hist}}_B)$$
$$V = \bigcup_{v \in \text{instances}(\text{Steuerungsvorgabe})} : \left[\bigcup_{p \in \text{prodregeln}(B,v)} : \{\text{explakt}(p)\}\right]$$

Zu beachten ist, dass die Produktionsregeln den Bestandteil hist einer Produktbibliothek nicht verändern können.

5.4 Integrationsmethodik für Prozessbeschreibungssprachen

Die Integrationsmethodik beschreibt das *Vorgehen* und die *Ergebnisstrukturen* zur Integration einer Prozessbeschreibungssprache mit dem Automatisierungskonzept. Diese werden im Folgenden formal definiert. Zur Anschauung wird das Beispiel aus Abbildung 5.14 (im Lösungsansatz) fortgeführt.

Definition 5.34: Integrationsmethodik
*Die **Integrationsmethodik** ist durch das UML-Aktivitätsdiagramm in* Abbildung 5.32 *definiert, sowie durch die Definitionen für die Zwischenergebnisse* PBSSEM *(Kapitel 5.4.1) und* BIBREFINE *(Kapitel 5.4.2). Auf die einzelnen Schritte wird in den nachfolgenden Definitionen eingegangen.*

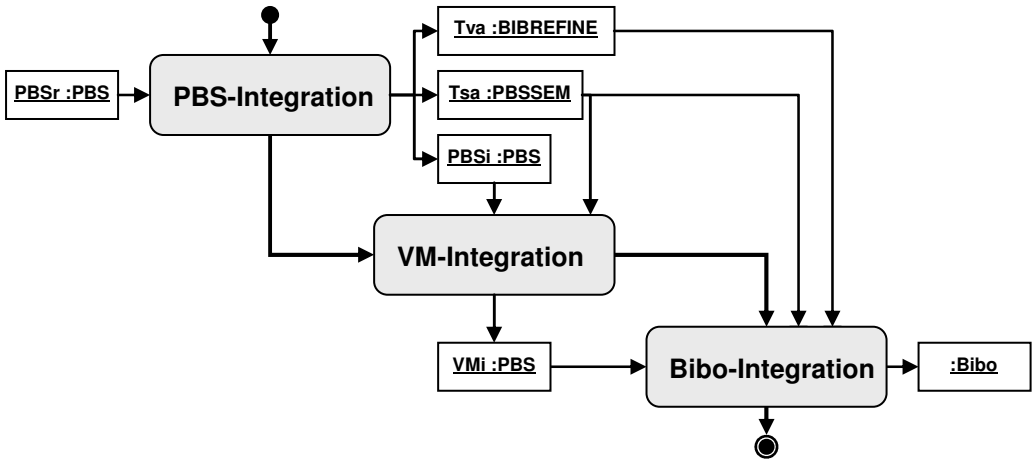


Abbildung 5.32 Integrationsmethodik

Obwohl die Schritte der Integrationsmethodik nach dieser Definition genau einmal in der vorgegebenen Reihenfolge zu durchlaufen sind, soll dieser Ablauf nicht als Beschränkung der Allgemeinheit verstanden werden. Vielmehr sind auch Rücksprünge erlaubt - zur Übersicht hier jedoch nicht explizit weiterverfolgt.

Definition 5.35: PBS-Integration
*Die **PBS-Integration** ist durch das UML-Aktivitätsdiagramm in* Abbildung 5.33 *definiert.*

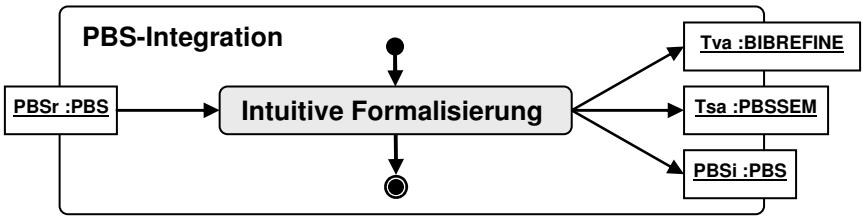


Abbildung 5.33 PBS-Integration

Die PBS-Integration kann als eine referenziell nicht transparante Funktion verstanden werden, die eine Prozessbeschreibungssprache als Eingabe nimmt, und eine neue, integrierte Prozessbeschreibungssprache, zusammen mit einer Verfeinerungsabbildung (Kapitel 5.4.1) und einer Semantikabbildung (Kapitel 5.4.2) als Ausgabe liefert. Die Gestaltung dieser Funktion kann im Allgemeinen nicht formal festgelegt werden, da einerseits beliebige Prozessbeschreibungssprachen integriert werden können sollen; andererseits eine Integration einer beliebig ausgewählten Prozessbeschreibungssprache von Informationen abhängt, die, realistisch gesehen, formal nicht greifbar sind. Beispielsweise können bei der Integration

getroffene Entscheidungen von Personen und deren eigenen Vorstellungen, Wünschen sowie Gewohnheiten abhängen. Um dennoch eine Hilfestellung zur Ausführung dieser Funktion zu geben, wird hier die Technik der intuitiven Formalisierung (Kapitel 5.4.3) verwendet – ohne dies als Beschränkung der Allgemeinheit zu verstehen.

Definition 5.36: VM-Integration
*Das **Integrieren eines Vorgehensmodells** mit dem Automatisierungskonzept entspricht dem Ausführen der folgenden referenziell nicht transparante Funktion:*

$$\forall\ pbs \in PBS, Tsa \in PBSSEM:$$
$$vmintegration(pbs,Tsa) =_{spec} vmi = any(\{\ vmi \in VM \mid instance<vmi,pbs>\ \})$$

Die VM-Integration nimmt eine Prozessbeschreibungssprache pbs und deren Semantik als Eingabe, und liefert ein dazu passendes Vorgehensmodell vmi als Ausgabe. Die Semantik dient als Entscheidungsgrundlage, da das Verwenden von Sprachkonstrukten aus pbs ohne der Kenntnis deren Wirkung nicht realitätsnah wäre. Darüber hinaus kann diese Funktion intern noch weitere Eingaben verwenden – wie beispielsweise ein ursprüngliches, nicht integriertes Vorgehensmodell, was es zu integrieren gilt. Die konkrete Gestaltung dieser Funktion kann jedoch – aus den gleichen Gründen wie bei der PBS-Integration – nicht formalisiert werden. Da das Aufstellen von Vorgehensmodellen anhand einer gegebenen Prozessbeschreibungssprache orthogonal zur Operationalisierung des Projektcontrollings steht, werden in dieser Arbeit keine konkreten Techniken vorgestellt.

Definition 5.37: Bibo-Integration
*Die **Bibo-Integration** beschreibt, wie die Ergebnisse einer angewendeten PBS-Integration im Rahmen des Automatisierungskonzepts auch tatsächlich eingesetzt werden. Im Unterschied zur Integrationsmethodik handelt es sich dabei um eine vollständig formalisierbare Funktion:*

$$bibointegration(Tva, Tsa, vmi) =_{def} pbssem(\ applybibrefine(\ Tva\ ), Tsa, vmi\ )$$

Die Bibo-Integration liefert eine Produktbibliothek deren internes Modell durch die Verfeinerungsabbildung Tva verfeinert wird, und deren Graph- und Produktionsregelanteile durch die Semantikabbildung Tva befüllt werden. Die dazu eingesetzten Funktionen applybibrefine und applypbsem werden in den Kapiteln 5.4.1 und 5.4.2 definiert.

5.4.1 Verfeinerungsabbildung

Eine Verfeinerungsabbildung beschreibt, eine Verfeinerung des Produktbibliotheksmodells. Zur einfacheren Handhabung muss die Verfeinerungsabbildung selbst noch nicht eine „echte“ Verfeinerung des Produktbibliotheksmodells sein. D.h. es müssen nicht alle Elemente des Produktbibliotheksmodells darin vorkommen, sondern nur diejenigen, die verfeinert werden sollen. Erst nach Anwendung der Verfeinerungsabbildung auf das Produktbibliotheksmodell muss eine „echte“ Verfeinerung entstehen.

Definition 5.38: Verfeinerungsabbildung
Die Menge PBSVER *aller **Verfeinerungsabbildungen** ist wie folgt definiert:*

$$PBSVER =_{def} \{\ Tva \in MODEL \mid refinement<Tva + Bibomodell, Bibomodell>\ \}$$

Die Verfeinerungsabbildung entspricht der Menge derjenigen Modelle, die, erst zusammen mit dem Produktbibliotheksmodell, eine Verfeinerung des Produktbibliotheksmodells ergibt.

Definition 5.39: Anwendung einer Verfeinerungsabbildung
*Die **Anwendung der Verfeinerungsabbildung** ist durch die Funktion* pbsver *als die Summierung mit dem Produktbibliotheksmodell definiert:*

$$bpsver(\ Tva\ ) =_{def} Bibomodell + Tva$$

5.4.2 Semantikabbildung

Die Semantikabbildung ist eine Transformation die ein Vorgehensmodell in Graphelemente und (virtualisierte) Produktionsregeln einer Produktbibliothek überführt. Da prinzipiell beliebige Prozessbeschreibungssprachen und Verfeinerungen des Produktbibliotheksmodells in Frage kommen, kann die Ein- und Ausgabe der Transformation dieselben Strukturen aufweisen. Im Beispiel der Abbildung 5.14 (Lösungsansatz) kommt der Graphelementtyp Produkt nach der Verfeinerung sowohl in dem durch die Semantik zu transformierenden Vorgehensmodell VMi als auch im Modell der Produktbibliothek vor, in die VMi überführt wird. Um diesbezüglich Eindeutigkeit zu schaffen, wird das Konzept der Domänen verwendet. Konkret werden dazu die zwei Domänen semin und semout für die Ein- bzw. Ausgabe vorgesehen.

Der Wertebereich aller Semantikabbildungen wird durch das Produktbibliotheksmodell und dessen mögliche Verfeinerungen geprägt. Dabei ist allerdings nicht das gesamte Produktbibliotheksmodell nutzbar, sondern, wie im Lösungsansatz in der Abbildung 5.17 gezeigt, nur das Teilmodell Strg. Der Wertebereich aller Semantikabbildungen ist folglich gegenüber dem Produktbibliotheksmodell auf die entsprechende Teilbibliothek beschränkt.

Definition 5.40: Semantikabbildung
*Eine **Semantikabbildung*** Tsa $\in$ PBSSEM *ist eine Transformation deren Produktionsregeln mindestens die Domänen* semin *und* semout *besitzen, und das Domänenmuster Instanz des Produktbibliotheksmodells, oder Verfeinerungen davon, ist*

$$PBSSEM =_{def} \bigcup m \in MODEL \text{ mit } refinement<m,Strg>: \{\ tr \in TRANS \mid [\forall\ p \in tr: \{semin,semout\} \subset WB(\underline{dz_p})] \wedge instance< p \restriction_G \underline{dom_p}(semout), m > \}$$

Definition 5.41: Anwenden einer Semantikabbildung
*Das **Anwenden einer Semantikabbildung** ist durch die Funktion* pbssem *die ein* Tsa $\in$ PBSSEM *und ein Vorgehensmodell* vgm $\in$ VM *als Eingabe nimmt, und eine Produktbibliothek* B'' $\in$ BIBO *als Ausgabe liefert, wie folgt definiert:*

$$pbssem(Tsa, vgm) =_{def} B'' = ändvirt(B,B',Strg) \text{ mit}$$
$$B' = B+g$$
$$B = Initialbibo$$
$$g = trans_1(\ vgm, Tsa, semin, semout\ )$$

5.4.3 Intuitive Formalisierung

Bei dem Aufstellen einer (Prozessbeschreibungs-)Sprache ist deren Mächtigkeit (im Sinne der Anzahl damit berechenbaren Funktionen) nicht das einzig entscheidende Kriterium. Beispielsweise erfüllt die Sprache der Turingprogramme dieses Kriterium in maximaler Weise, dennoch haben sich in der Praxis „höhere" Sprachen entwickelt die von den für das jeweilige Einsatzgebiet irrelevanten Details abstrahieren, und so deren Anwendung vereinfachen. Beispielsweise abstrahiert die Programmiersprache C durch Konstrukte wie *if* oder *while* vom Befehlszähler (vgl. [93]), und Java durch den Garbagecollector von der

Speicherbereinigung. Werden diese Überlegungen fortgeführt, ist festzustellen, dass auch Programmbibliotheken und selbst ganze SW-Programme ebenfalls (und als noch höhere) Sprachen zu verstehen sind: Sie bieten einem Anwender durch ihre Anwendungs- bzw. Benutzungsschnittstelle sehr spezialisierte Konstrukte (in Form von aufrufbaren Funktionen), die von noch mehr Details abstrahieren. Die vorgenommenen Abstraktionen können dabei auf Kosten der Mächtigkeit gehen. Dies ist jedoch nicht zwingend ein Nachteil, sondern kann, im Gegenteil, die Anwendung vereinfachen und damit einen Vorteil bieten.

Die Rolle des Prozessingenieurs ist in dem aufgestellten Spektrum der Sprachen eher an dem Ende der SW-Programme anzusiedeln. In der Praxis beschäftigt sich ein Prozessingenieur typischer weise nicht mit der Programmierung, so dass für die integrierte Prozessbeschreibungssprache möglichst einfache, selbsterklärende Sprachkonstrukte nötig sind. Die Komplexität und Vielfalt der für die Praxis möglicherweise relevanten Vorgaben erlaubt es jedoch nicht, eine einfache, für alle Vorgehensmodelle anwendbare Prozessbeschreibungssprache zu gestalten.

Die intuitive Formalisierung versteht daher die Integration einer Prozessbeschreibungssprache als Entwicklungsprojekt: Die für den jeweiligen Kontext relevanten Konstrukte werden – wie bei der Entwicklung einer Benutzungsschnittstelle für ein SW-Programm (vgl. [94]) auch – durch eine Anforderungsanalyse ermittelt und durch die Implementierung mit einer Semantik unterlegt. Prinzipiell sind damit alle Techniken einsetzbar, die auch für SW-Entwicklung nutzbar sind. Für das spezielle Ziel einer Prozessbeschreibungssprache die sich in das Automatisierungskonzept integriert, werden jedoch von der intuitiven Formalisierung spezielle Methoden für ausgewählte Schritte einer solchen Entwicklung zur Verfügung gestellt.

Definition 5.42: Intuitive Formalisierung
*Die **Intuitive Formalisierung** ist durch das UML-Aktivitätsdiagramm in* Abbildung 5.34 *definiert, sowie durch die Definitionen der zugehörigen Zwischenergebnisse und der einzelnen Schritte.*

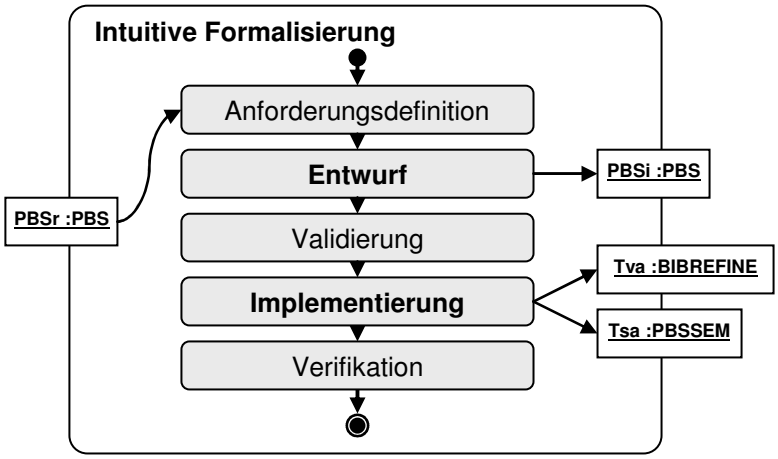


Abbildung 5.34 Intuitive Formalisierung

Ohne Beschränkung der Allgemeinheit gliedert die intuitive Formalisierung die Entwicklung einer Prozessbeschreibungssprache in die folgenden fünf Schritte:

- **Anforderungsdefinition:** In der Anforderungsdefinition (vgl. [95]) wird zusammengetragen welche Art von Vorgaben („Use Cases“) ein Prozessingenieur definieren können soll, die in der zu integrierenden Prozessbeschreibungssprache noch nicht mit geeigneten Konstrukten formalisiert sind. Abbildung 5.35 zeigt dazu ein einfaches Beispiel was im Weiteren fortgeführt wird.

- **Entwurf:** Im Entwurf wird die Erweiterung der Syntax der zu integrierenden Prozessbeschreibungssprache aus den Use Cases abgeleitet. Die Erweiterung stellt das erste Ergebnis der intuitiven Formalisierung dar – die Syntax der vom Prozessingenieur anzuwendenden, integrierten Prozessbeschreibungssprache.
- **Validierung:** Anhand von Anwendungsbeispielen ist die Mächtigkeit und Intuitivität der Anwendung der Sprachkonstrukte zu untersuchen.
- **Implementierung:** In der Implementierung werden die Konstrukte der integrierten Prozessbeschreibungssprache durch das Aufstellen einer Semantikabbildung mit einer formalen Semantik unterlegt. Sofern sich dazu eine Verfeinerung des Produktbibliotheksmodells eignet, wird diese in die Verfeinerungsabbildung aufgenommen.
- **Verifikation:** Zur Verifikation der Semantikdefinition wird diese auf die Anwendungsbeispiele angewendet. Die resultierenden Produktionsregeln werden hinsichtlich ihrer Wirkung mit den Use Cases verglichen.

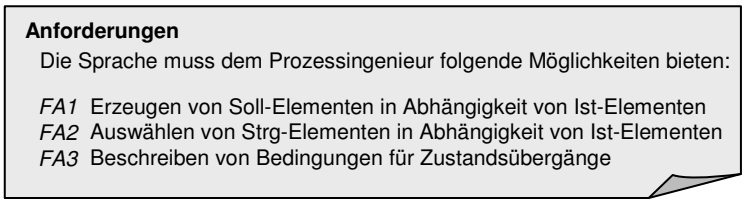


Abbildung 5.35 Anforderungen an die Prozessbeschreibungssprache

Sofern bei einem der Schritte Unstimmigkeiten gefunden werden, kann in einen früheren Schritt zurückgekehrt werden. In Abbildung 5.34 werden diese Rücksprungmöglichkeiten zur Übersicht nicht explizit dargestellt.

Für die Schritte Entwurf und die Implementierung werden in den Kapiteln 5.4.3.1 und 5.4.3.2 jeweils konkrete Techniken bzw. Methoden zur Verfügung gestellt.

5.4.3.1 Entwurf

Beim Entwurf werden aus den Use Cases Schemata für Gefügeketten erarbeitet, die der Grammatik der natürlichen Sprache folgen. Dabei sollte die Reihenfolge der Satzbausteine eines Use Cases bei der Abbildung auf Sprachkonstrukte in der Tendenz erhalten bleiben. Dadurch kann die Semantik einer in dieser Struktur abgelegten Vorgabe intuitiv erfasst werden (woraus sich auch die Namensgebung der intuitiven Formalisierung ergibt). Um die Anwendung der Sprachkonstrukte intuitiv zu halten sollten ähnliche Konzepte auch mit ähnlichen Konstrukten bzw. Strukturen abgebildet werden. Beispielsweise sollte das Filtern von Ist-Inhalten nach einem bestimmten Inhaltstyp ein ähnliches lautendes Konstrukt wie das Filtern von Ist-Ergebnissen nach einem bestimmten Ergebnistyp sein. Weiterhin sollten Konstrukte möglichst kontextfrei sein, die Semantik einer Vorgabe sich also „summarisch“ zusammensetzen.

Notiert werden die Schemata für Gefügeketten mit UML-Aktivitätsdiagrammen. Da UML-Aktivitätsdiagramme formal noch keine Prozessbeschreibungssprache sind, wird die Funktion *umltrans* bereit gestellt, mit der aus UML-Aktivitätsdiagrammen eine (Erweiterung für eine) Prozessbeschreibungssprache berechnet werden kann.

Für den Einsatz einer solchen Prozessbeschreibungssprache wird zusätzlich das Konzept von Phrasen eingeführt, die als Abkürzungen für längere bzw. häufig genutzte Ketten wirken.

Definition 5.43: UML-Aktivitätsdiagramm-Modell
*Das **UML-Aktivitätsdiagramm-Modell*** UMLADModell ∈ MODEL *ist durch* Abbildung 5.36 *definiert.*

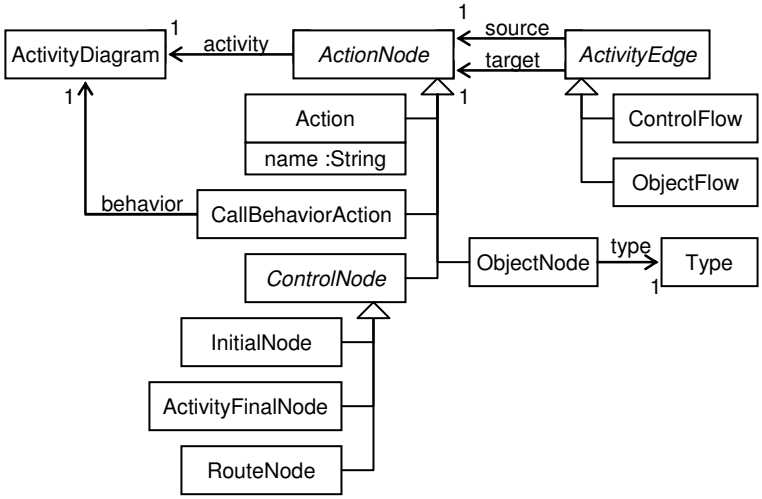


Abbildung 5.36 Das UML-Aktivitätsdiagramm-Modell

Das UML-Aktivitätsdiagramm-Modell ist der für diese Arbeit relevante Ausschnitt aus der Definition der UML-Aktivitätsdiagramme in [47] mit folgenden Abweichungen:

- Die CallBehaviorAction erbt nicht von Action, sondern von ActionNode. Dies vereinfacht die Definition der Transformationsfunktion, da bei Produktionsregeln die nur für Actions, aber nicht für CallBehaviorActions gelten sollen, nicht zusätlich auf die exakte Typübereinstimmung geprüft werden muss. Der Name der CallBehaviorAction ist zudem nicht weiter von Interesse und wird bei der Transformation nicht verarbeitet.
- Das Konstrukt ObjectNode erbt nicht von TypedElement, sondern erhält direkt die Assoziation type. Dies erlaubt es auf die Einführung und Verwendung von Mehrfachvererbung zu verzichten.
- Die Containment-Beziehung von ActivityDiagram zu ActionNode wird durch eine 1:N-Assoziation ersetzt, die jedem ActionNode genau ein ActivityDiagram zuordnet. Dies erlaubt es auf die Einführung und Verwendung von Containment-Beziehungen zu verzichten.
- Zu Objektknoten dürfen Kanten nur hinführen. Aus Objektknoten abführende Kanten sind im Rahmen der Sprachableitung nicht definiert.
- Es wird nicht zwischen MergeNode und DecisionNode unterschieden. Stattdessen werden beide als RouteNodes geführt. Dies verkürzt die Notationen, wie in Abbildung 5.37 dargestellt.

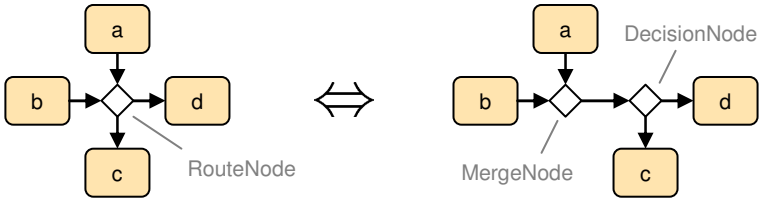


Abbildung 5.37 Notation von Merge- und DecisionNodes als RouteNodes

Unveränderte Constraints, wie beispielsweise die Forderung, dass InitialNodes nur abführende und ActivityFinalNodes nur eingehende Kanten haben dürfen, gelten auch hier – sind jedoch nicht explizit angegeben.

Definition 5.44: UML-Aktivitätsdiagramm
Die Menge UMLAD *aller* ***UML-Aktivitätsdiagramme*** *sei wie folgt definiert:*

UMLAD = { g∈GRAPH | instance<g,UMLADModell> }

Ein Ad ∈ UMLAD kann eine Menge von (ggf. zusammenhängenden) UML-Aktivitätsdiagrammen enthalten. Begrifflich wird jedoch im Folgenden auch bei einem solchen Ad von einem Aktivitätsdiagramm gesprochen. Abbildung 5.38 zeigt hierzu ein Beispiel, mit vier Aktivitätsdiagrammen (Knoten mit den Ids d1 – d4). Ebenfalls dargestellt sind die Ids der darin enthaltenen ActivityNodes und ActivityEdges.

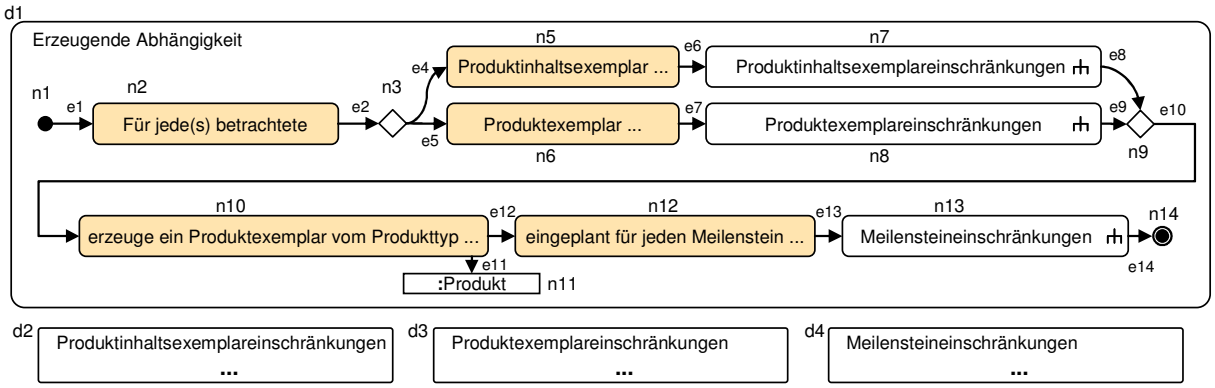


Abbildung 5.38 Beispiel eines zu tranformierenden UML-Aktivitätsdiagramms

Definition 5.45: Gefügemodell
Das ***Gefügemodell*** Gefügemodell ∈ MODEL *ist durch die* Abbildung 5.39 *definiert.*

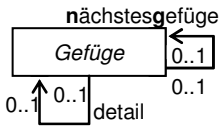


Abbildung 5.39 Gefügemodell

Das Gefügemodell beschreibt den grundlegenden Teil der bei der Transformation erzeugten Prozessbeschreibungssprache: sämtliche Terminale werden über Vererbung vom Konstrukt Gefüge eingeführt, wodurch jede erzeugte Prozessbeschreibungssprache eine Verfeinerung des Gefügemodells ist.

Definition 5.46: Modell für graphbasierte, hierarchische Textgrammatiken
Das ***Modell für graphbasierte, hierarchische Textgrammatiken*** Textgrammatikmodell $\in$ MODEL *ist durch die* Abbildung 5.40 *definiert.*

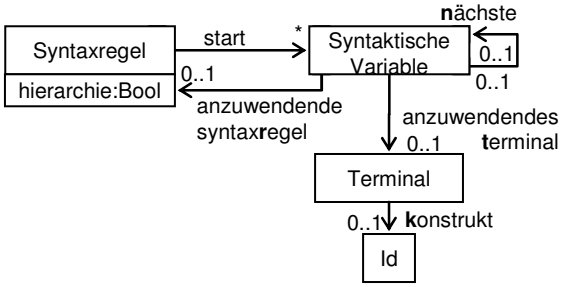


Abbildung 5.40 Modell für graphbasierte, hierarchische Textgrammatiken

Das Modell der graphbasierten, hierarchischen Textgrammatik beschreibt die Struktur des Zwischenergebnisses bei der Transformation eines UML-Aktivitätsdiagramms in eine das Gefügemodell verfeinernde Prozessbeschreibungssprache. Im Hinblick auf die Grundlagen stellt dieses Textgrammatikmodell ein vereinfachtes Virtualisierungsmodell für Graphgrammtiken dar. Vereinfacht bedeutet, dass als Wörter nur noch solche Graphen beschreibbar sind, die (hierarchische) Ketten bilden. Für die Bezeichnung der Konstrukte wird die Begrifflichkeit der textbasierten Grammatiken aus der theoretischen Informatik [33] wie folgt aufgegriffen:

- Terminal und syntaktische Variable entsprechen genau den gleichen Bezeichnungen in textbasierten Grammatiken. Eine Produktionsregel in textbasierten Grammatiken wird hier, um Verwechslungen zu vermeiden, als Syntaxregel bezeichnet.
- Durch die Assoziation start wird das erste Element der „rechten Seite“ einer Syntaxregel bestimmt. Alle weiteren Elemente ergeben sich durch die Assoziation nächste.
- Eine Syntaxregel kann mehrere „rechte Seiten“ haben, was bei textbasierten Grammatiken mehreren Syntaxregeln mit denselben „linken Seite“ entspricht.
- Eine Syntaxregel ohne „rechten Seite“ bezeichnet hier eine Syntaxregel die das leere Wort, also einen leeren Graph, erzeugt.
- Der Zusammenhang zwischen einer syntaktischen Variable und der dahinterstehenden Syntaxregel wird durch die Assoziation anzuwendende syntaxregel festgelegt.

Im Unterschied zu textbasierten Grammatiken

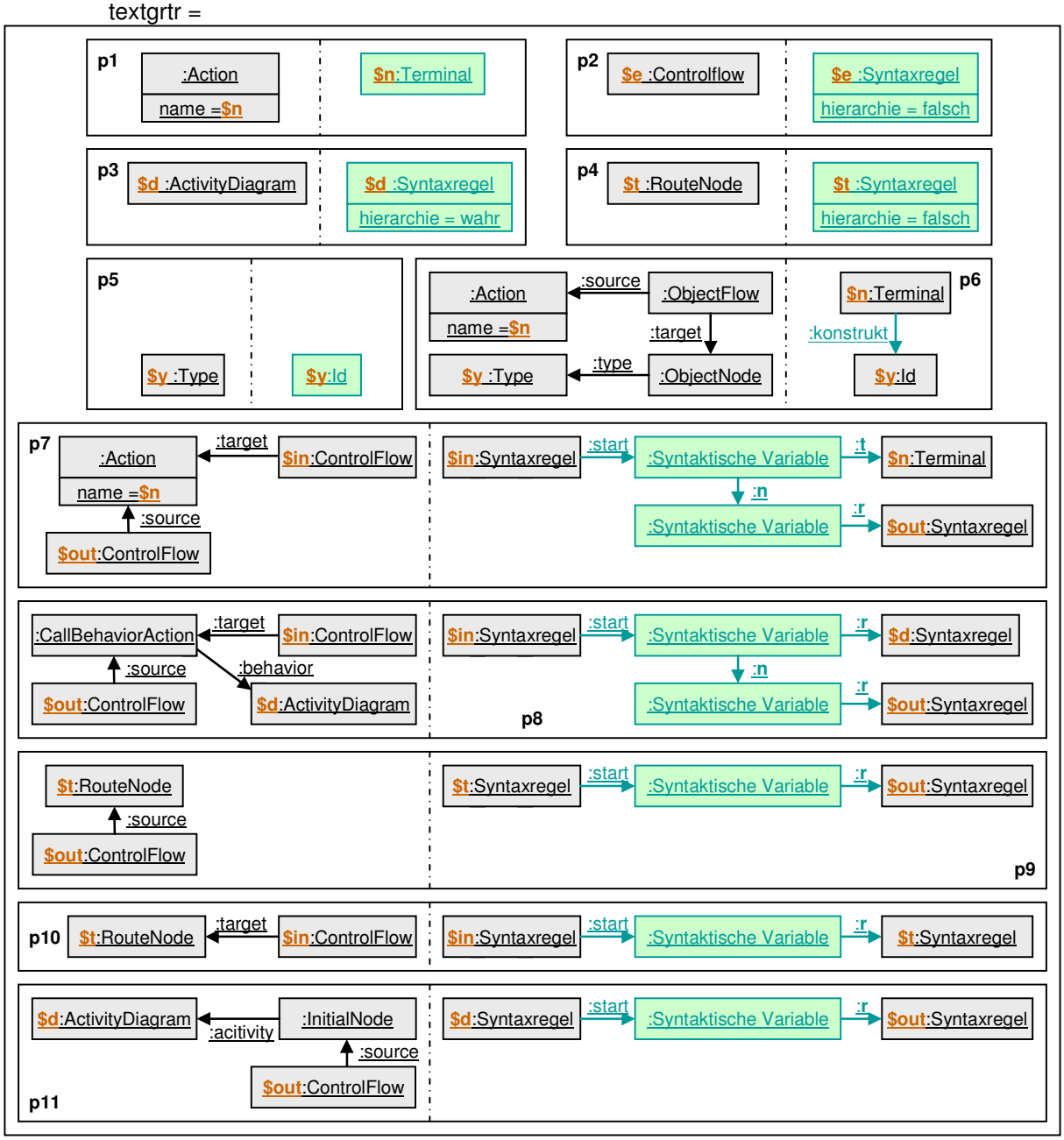
- sind Elemente der „rechten Seite“ hier ausschließlich syntaktische Variablen, wobei eine syntaktische Variable selbst auf (höchstens) ein Terminal über die Assoziation anzuwendendes terminal abbilden kann. Dies vereinfacht lediglich die Definition der Transformationsabbildung und stellt keine Einschränkung der Mächtigkeit dar.
- können Syntaxregeln hierarchisch sein. Eine hierarchische Syntaxregel besagt dass die durch sie erzeugte Gefügekette als Detail und nicht als Verlängerung der bisherig soweit erzeugten anzuhängen ist.
- können Terminale über die Assoziation konstrukt eine Id referenzieren. Damit können Terminale mit (nicht-Gefüge-)Konstrukten aus der Prozessbeschreibungssprache parametrisiert werden. Ein Wort, welches ein Terminal instanziert, kann dann eine Instanz des Konstruktes referenzieren. Beispielsweise das Produkt Systemarchitektur, wenn das entsprechende Terminal über die Assoziation konstrukt auf das Konstrukt Produkt verweist.

Definition 5.47: graphbasierte, hierarchische Textgrammatiken
Die Menge GHT *aller* ***graphbasierten, hierarchischen Textgrammatiken****, kurz Textgrammatiken, ist wie folgt definiert:*

GHT = { g ∈ GRAPH | instance<g,Textgrammatikmodell> }

Definition 5.48: Transformation von UML-Aktivitätsdiagrammen zu Textgrammatiken
Die Funktion textgrtrans ***transformiert*** *ein UML-Aktivitätsdiagramm* Ad ∈ UMLAD *in eine Textgrammatik* ght ∈ GHT *um, und ist wie folgt definiert:*

textgrtrans(Ad) $=_{def}$ ght = $trans_1$(Ad, textgrtr, input, output)
wobei
textgrtr =



Die Transformation besteht von der Idee her darin, die Knoten eines Aktivitätsdiagramms auf Terminale und die Kanten auf Syntaxregeln abzubilden. Da Terminale mehrfach verwendet werden, können mögliche Verknüpfungen nicht direkt auf Terminalen definiert werden.

Stattdessen ist ein Hilfskonstrukt notwendig – diese Aufgabe wird von den Syntaktischen Variablen übernommen. Weiterhin stellen nicht alle Knoten eines Aktivitätsdiagramms Terminale dar – sondern nur die Actions. Andere ActivityNodes und ActivityDiagramme selbst haben ebenfalls eine steuernde Wirkung bezüglich der gültigen Wörter und werden daher ebenfalls auf Syntaxregeln abgebildet. Bei den Kanten ist zwischen ControlFlow und ObjectFlow zu unterscheiden. Letzteres beschreibt nur die Verbindung eines einzelnen Terminals zu einem anderen (nicht-Gefüge) Konstrukt der Prozessbeschreibungssprache und hat damit keine Auswirkung auf die Verkettungsmöglichkeiten der Gefüge untereinander.

Im Einzelnen gestalten sich die bei der Transformation eingesetzten Produktionsregeln wie folgt:

- p1: Der Name jeder Action bildet die Id eines Terminals.
- p2: Jeder Kontrollfluss bildet eine nicht-hierarchische Syntaxregel.
- p3: Jedes UML-Aktivitätsdiagramm bildet eine hierarchische Syntaxregel.
- p4: Jeder RouteNode bildet eine nicht-hierarchische Syntaxregel.
- p5: Jeder Typ wird auf eine Id abgebildet, der ein (nicht-Gefüge-)Konstrukt der Prozessbeschreibungssprache repräsentiert.
- p6: Jeder Objektfluss bildet eine Kante zwischen Terminal und entsprechendem Konstrukt.
- p7: Erzeugt die „rechte Seite“ für eine in eine Action eingehende Kante. Die „rechte Seite“ besteht aus zwei syntaktischen Variablen: Eine die das Terminal der Action erzeugt, und eines die die nächste Syntaxregel aufruft – welche sich durch die aus der Action abführenden Kante bestimmt.
- p8: Erzeugt die „rechte Seite“ für eine in eine CallBehaviorAction eingehende Kante. Im Unterschied zu p7 wird anstelle des Terminals die Syntaxregel für das verknüpfte Aktivitätsdiagramm aufgerufen.
- p9: Jede von einem RouteNode abführende Kante erzeugt eine „rechte Seite“ für die zum RouteNode gehörende Syntaxregel.
- p10: Erzeugt die „rechte Seite“ für eine in einen RouteNode eingehende Kante. Diese besteht lediglich aus dem Aufruf der zum RouteNode gehörenden Syntaxregel.
- p11: Zu jedem InitialNode eines UML-Aktivitätsdiagramms wird eine „rechte Seite“ für die zum Aktivitätsdiagramm gehörende Syntaxregel erzeugt.

Abbildung 5.41 zeigt das Transformationsergebnis für den in Abbildung 5.38 gezeigten UML-Aktivitätsdiagramm.

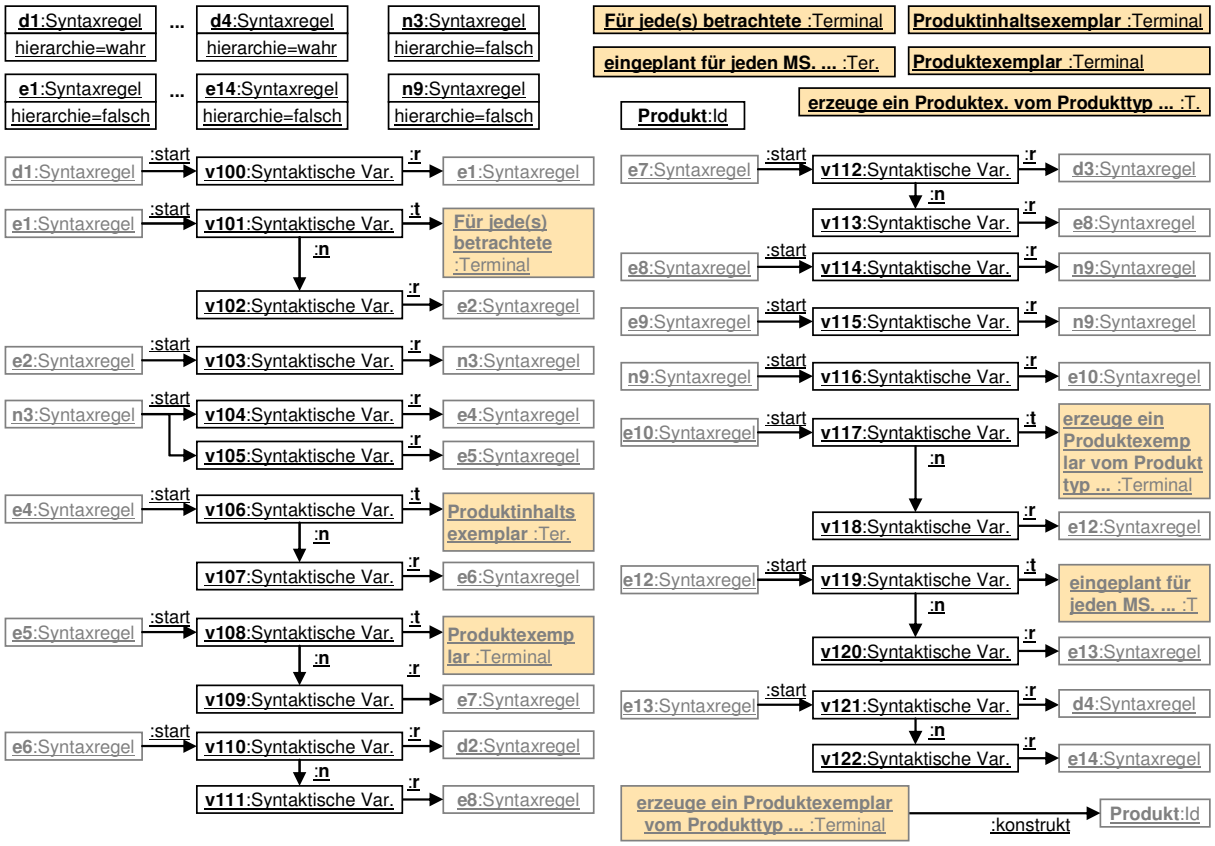
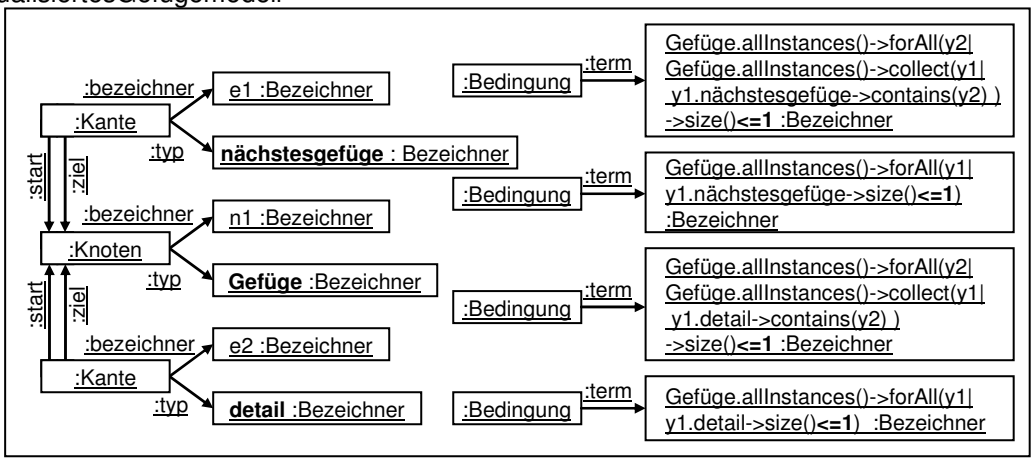


Abbildung 5.41 Transformation eines UML-Aktivitätsdiagramms in eine Textgrammatik

Definition 5.49: Transformation von Textgrammatiken zu Modellen

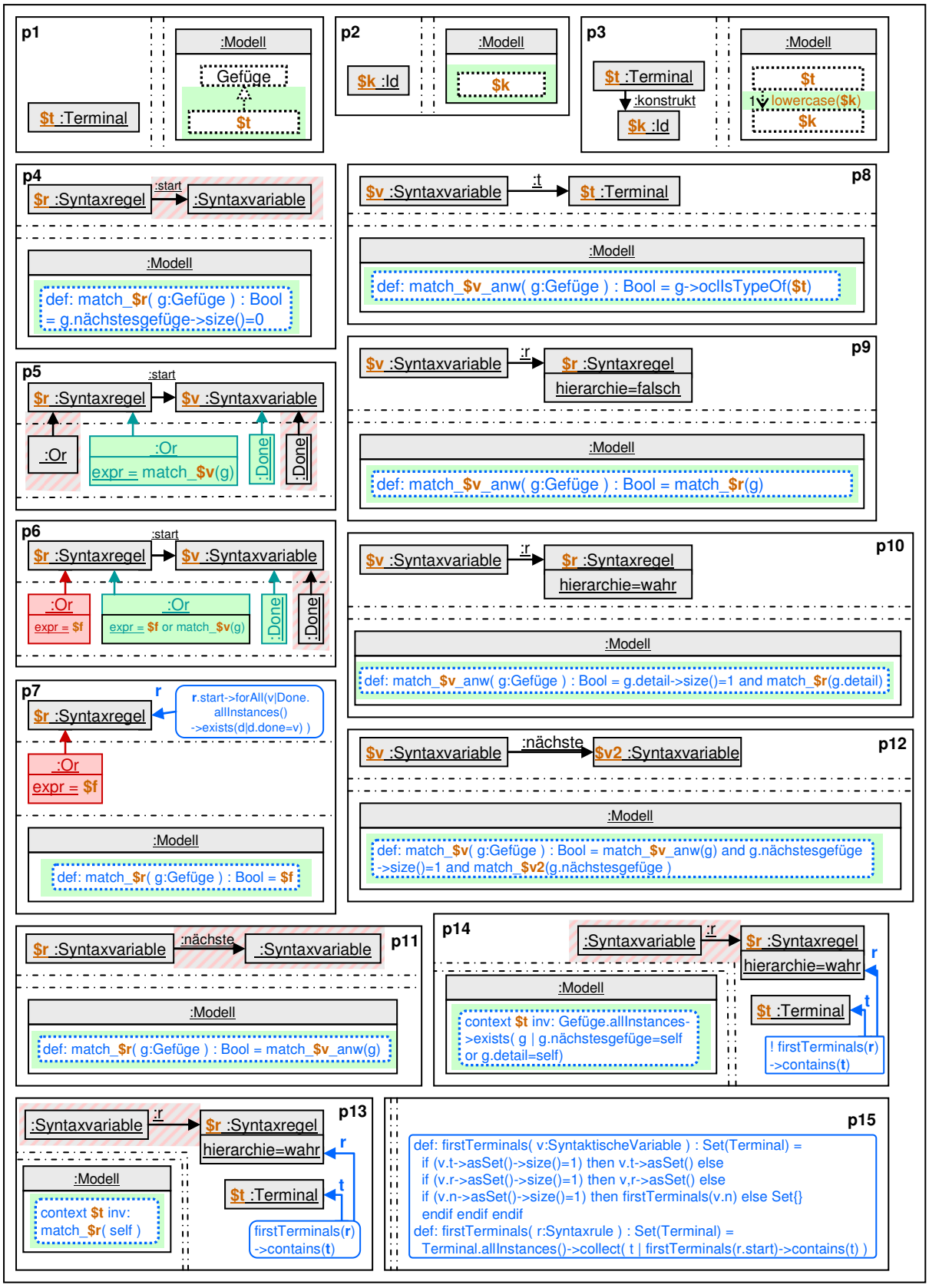
Die Funktion gefügetrans ***transformiert*** *eine Textgrammatik* ght ∈ GHT *in eine Prozessbeschreibungssprache* pbs ∈ PBS *um, und ist wie folgt definiert:*

gefügetrans(ght) $=_{def}$ pbs = devirtualizemod(g)
wobei
g = trans$_1$(ght+VirtualisiertesGefügemodell, gefügetr, input, output),
VirtualisiertesGefügemodell =



,

gefügetr =



Die Transformation besteht von der Idee her darin, die Terminale einer Textgrammatik auf neue Konstrukte (der somit entstehenden Prozessbeschreibungssprache) abzubilden, die von Gefüge erben. Abbildung 5.42 zeigt dazu das Schema der Transformation.

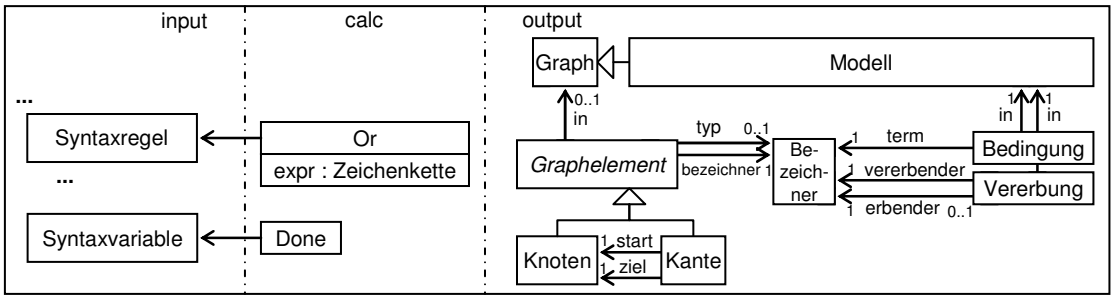


Abbildung 5.42 Transformationsschema für Transformation zu Gefüge

Durch das Gefügemodell sind Gefüge zunächst beliebig miteinander kombinierbar. Um die Kombinationen einzuschränken werden durch die Transformation entsprechende OCL-Constraints generiert. Für eine einfachere Transformation wird zu jeder Syntaxregel und jeder syntaktischen Variable eine (oder mehrere) zugehörige OCL-Funktionen erzeugt die sich gemäß der durch die „rechten Seiten" definierten Verbindungen gegenseitig aufrufen. Den Startpunkt der für die Strukturprüfungen bilden die jeweils ersten Terminale der obersten Syntaxregeln. Im Einzelnen gestalten sich die bei der Transformation eingesetzten Produktionsregeln wie folgt:

- p1: Jedes Terminal bildet ein das Konstrukt Gefüge spezialisierendes neues Konstrukt.
- p2: Sofern ein Terminal ein (nicht-Gefüge-)Konstrukt referenziert, bildet dies eine entsprechende Assoziationen.
- p3: Jedes (nicht-Gefüge-)Konstrukt erzeugt ein entsprechendes Konstrukt in der resultierenden Sprache. Diese Produktionsregel dient lediglich als Hilfestellung, sofern die Zielsprache nicht als Eingabe an die Transformation gegeben wird, sondern erst hinterher mit dem Transformationsergebnis vereinigt wird.
- p4: Sofern eine Syntaxregel keine „rechte Seite" hat, muss das geprüfte Gefüge am Ende sein, d.h. keine nächsten Gefüge mehr haben.
- p5-p7: Sofern eine Syntaxregel „rechte Seiten" hat, muss das geprüfte Gefüge auf mindestens eines dieser Seiten passen. Die Oder-Verknüpfung wird mit Hilfe der Produktionsregeln p5 und p6 gebildet. Die Produktionsregel p7 erzeugt Überführt die Verknüpfung in eine OCL-Funktion. Für die Prüfung der „rechte Seite" wird eine entsprechende Funktion aufgerufen die durch die Produktionsregeln p11 und/oder p12 erzeugt werden.
- p8: Sofern ein Gefüge durch eine Syntaktische Variable gegen ein Terminal geprüft wird, muss dieses vom entsprechenden Typ sein.
- p9-p10: Sofern ein Gefüge durch eine Syntaktische Variable gegen eine Syntaxregel geprüft wird, wird die Prüfung and die dafür zuständige OCL-Funktion weitergeleitet. Bei nicht-hierarchischen Syntaxregeln wird das aktuelle Gefüge übergeben, bei hierarchischen ist es das als Detail angehängte Gefüge.
- p11-p12: Sofern eine syntaktische Variable keinen Nachfolger, und damit das letzte Element in einer „rechten Seite" ist, wird nur die Prüfung des Gefüges bezüglich der syntaktischen Variable selbst vorgenommen (siehe Produktionsregeln p8-p10). Andernfalls wird zusätzlich auch die Prüfung der nachfolgenden Syntaktische Variable vorgenommen.
- p13: Betrachtet werden hierarchische Syntaxregel, die nicht selbst aufgerufen werden und somit implizit als eine Startregeln zu betrachten sind. Jedes von einer solche Syntaxregel aus gesehen erste Terminal bildet ein OCL-Constraint mit dem Terminal als OCL-Kontext und der OCL-Funktion der entsprechenden Syntaxregel als die zu geltende Bedingung.

- p14: Jedes Terminal welches nicht von p13 verarbeitet wird, darf kein erstes Gefüge in einer Gefügekette sein – muss also entweder einen Vorgänger haben, oder ein anderes Gefüge detaillieren.
- p15: Diese Regel liefert für p13 und p14 die OCL-Funktion zur Berechnung der ersten Terminale einer Syntaxregel.

Abbildung 5.43 skizziert das Transformationsergebnis für die in Abbildung 5.41 gezeigte Textgrammatik. Zur Übersicht werden die erzeugten OCL-Constraints nach den sie erzeugenden Produktionsregeln gruppiert dargestellt.

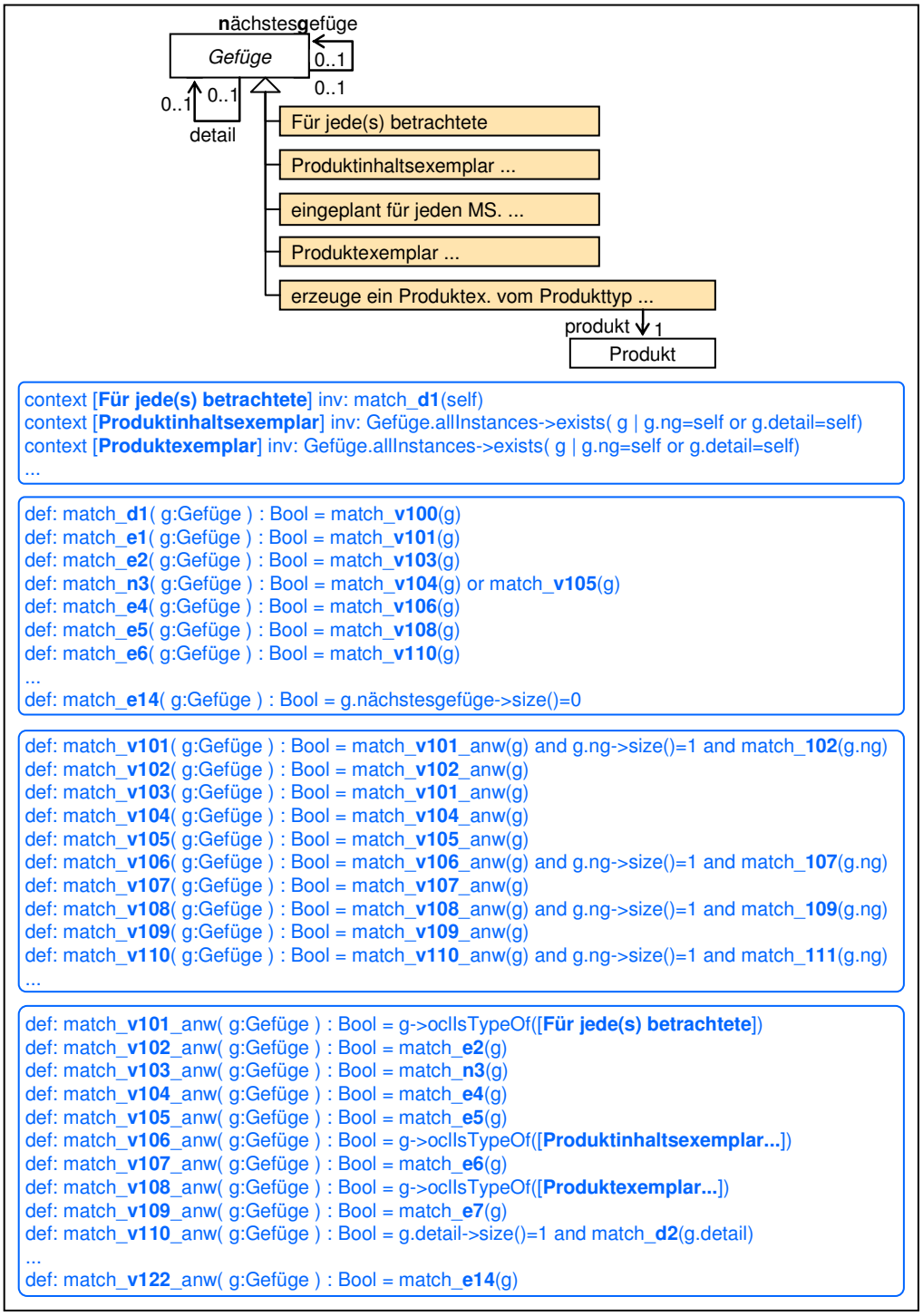
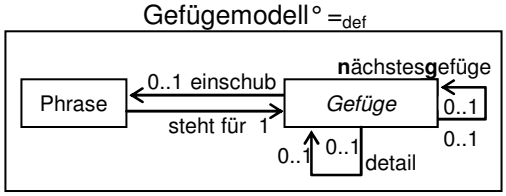


Abbildung 5.43 Beispiel des Ergebnisses nach der zweiten Transformationsstufe

Definition 5.50: Erweitertes Gefügemodell für Phrasen
Für das Konzept der ***Phrasen*** *ist die Definition des* Initialgefügemodells *wie folgt anzupassen:*



Das Konzept der Phrasen erlaubt es, Abkürzungen für Gefügeketten-Teile zu definieren und diese beliebig oft wiederzuverwenden. Eine Phrase steht für eine Gefügekette, die über die gleichnamige Assoziation definiert wird. Zur Verwendung einer Phrase kann ein beliebiges Gefüge eine Phrase einschieben, die dann unmittelbar vor dem jeweils nächsten Gefüge vollständig einzusetzen ist. Abbildung 5.44 zeigt dazu ein Beispiel.

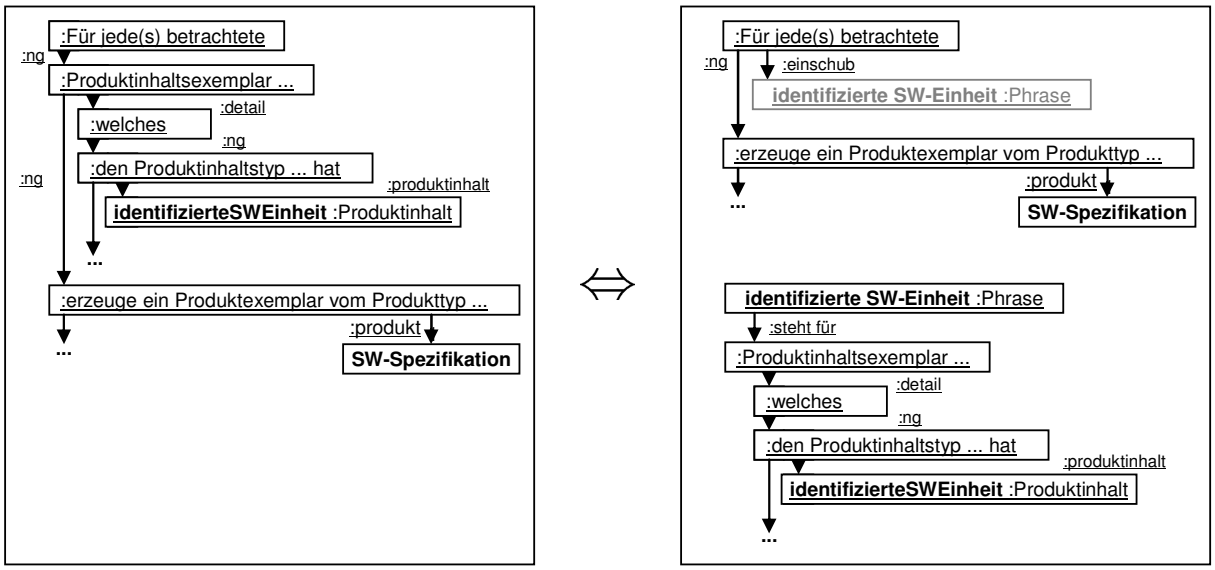


Abbildung 5.44 Beispiel zur Anwendung von Phrasen

Zu beachten ist, dass Phrasen lediglich als syntaktischer Zucker (vgl. [53]) wirken, d.h. keine Ausdrucksmächtigkeit hinzufügen. Insbesondere darf das Einschieben von Phrasen keine Zyklen bilden.

Definition 5.51: Ableitung einer Sprache aus einem UML-Aktivitätsdiagramm
Die Funktion umltrans *transformiert ein* Ad ∈ UMLAD *in eine Prozessbeschreibungssprache* pbs ∈ PBS *und ist wie folgt definiert:*

$$umltrans(Ad) =_{def} pbs = gefügetrans(textgrtrans(Ad))$$

5.4.3.2 Implementierung

Bei der Implementierung wird die im Entwurf aufgestellte Sprache mit einer Semantik unterlegt. Anhand eines Beispiels wird dazu im Folgenden eine Methodik vorgestellt. Konkret wird das Beispiel aus Abbildung 5.38 aufgegriffen, und um ein UML-Aktivitätsdiagramm für die CallBehaviorAction Produktinhaltsexemplareinschränkungen ausgestaltet. Dadurch kann der Umgang mit Hierarchien behandelt werden. Ein dazu konstruierbares Wort ist in Abbildung 5.45 unten-rechts angegeben.

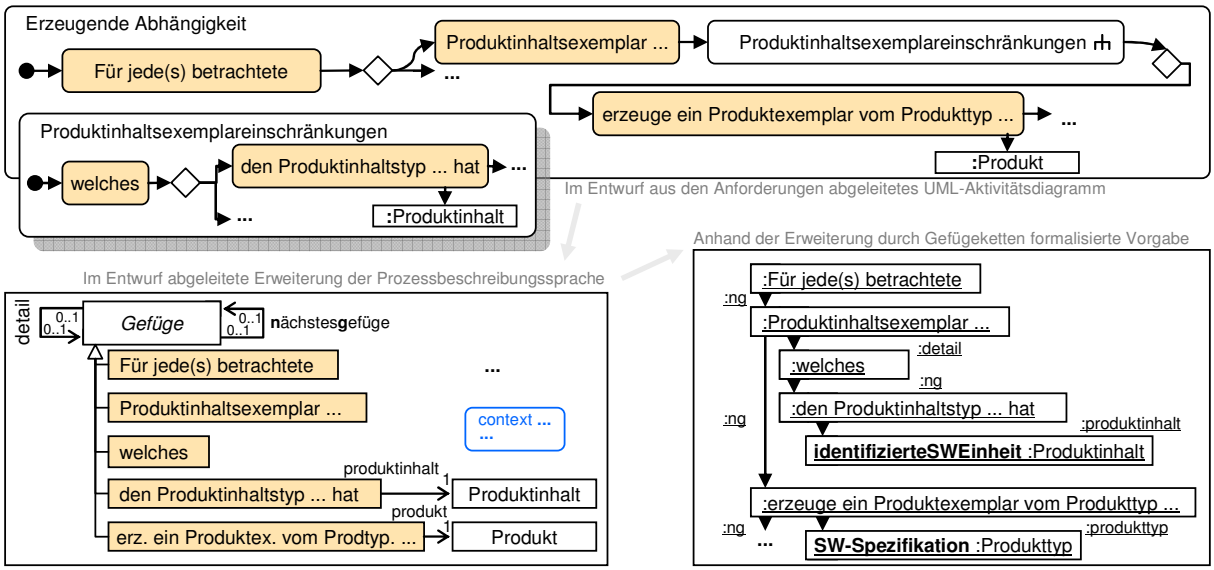


Abbildung 5.45 Beispiel zur Veranschaulichung der Implementierungs-Methodik

Die bei der Implementierung aufzustellende Semantikabbildung besitzt laut Definition (siehe Kapitel 5.4.2) mindestens die beiden Domänen semin und semout. Erstere enthält bei der Ausführung der Semantikabbildung die erweiterte Prozessbeschreibungssprache (vgl. Abbildung 5.45, unten-links). Letztere die entstehenden, virtualisierten Produktionsregeln. Als Hilfestellung wird noch eine dritte Domäne, cm, verwendet (vgl. Abbildung 5.46). Instanzen des Konstrukts Bezug geben an, zu welchen Gefügen die durch die Semantikabbildung erzeugten virtualisierten Produktionsregeln (bzw. Knoten davon) „gehören“. Diese Bezüge erleichern das Verarbeiten der Hierarchien in den Gefügeketten.

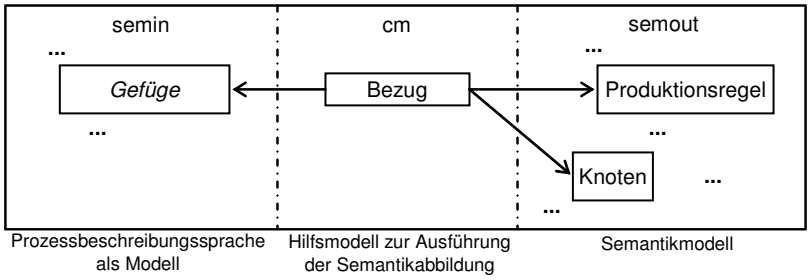


Abbildung 5.46 Eingesetzte Domänen bei Ausführung einer Semantikabbildung

Als eine weitere Erleichterung können die in Abbildung 5.47 definierten Notationen verwendet werden. Die erste kürzt die Verwendung eines Bezug-Musterknotens ab. Die zweite stellt die Bedingung, dass zwei Gefüge bezüglich der Hierarchie auf einer „Linie“ liegen, auf anschauliche Weise dar. Dabei sind nur Kanten vom Typ detail als Wechsel in der Hierarchie zu verstehen. Die durch die dritte Notation abgekürzte Bedingung fordert, dass das detaillierte Gefüge der unmittelbare „Vorgesetzte“ des detaillierenden ist. Die vierte Bedingung fordert, dass das eine Gefüge (identifiziert durch Suchmusterknoten n91) der erste

Vorgänger (dieses Typs) vom zweiten (identifiziert durch Suchmusterknoten n92) in der selben Hierarchieebene ist. Weitere Notationen können je nach Bedarf definiert werden.

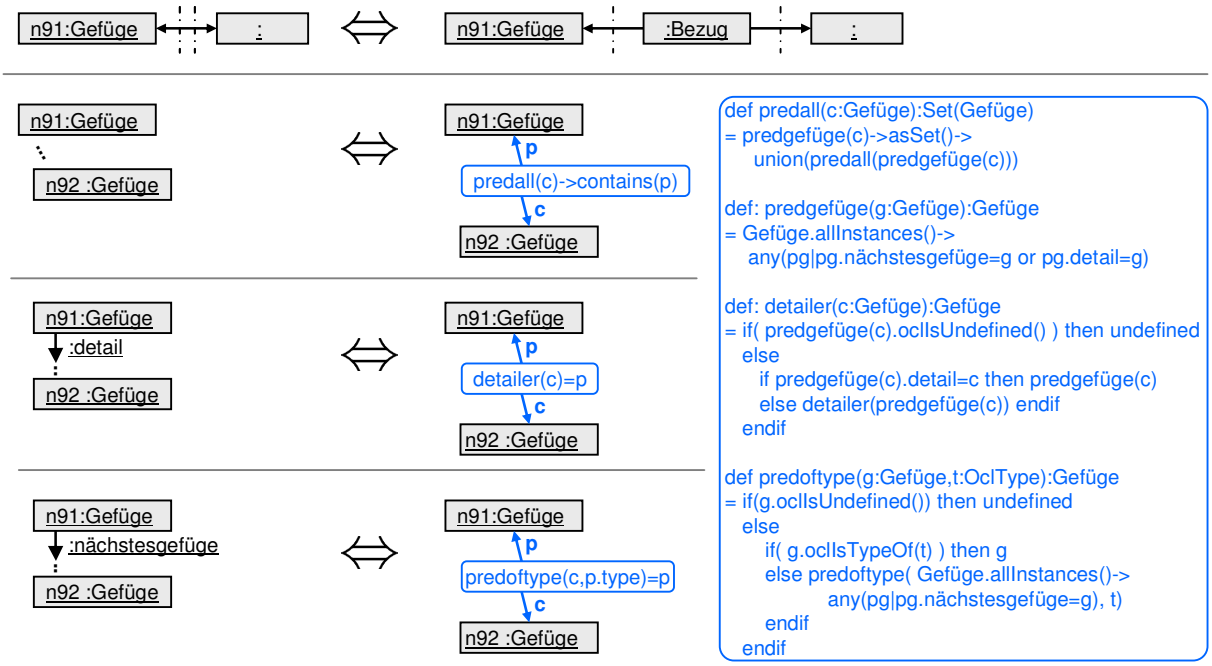


Abbildung 5.47 Notationen für Produktionsregeln einer Abbildungssemantik

Abbildung 5.48 zeigt eine Semantikabbildung für das laufende Beispiel. Dazu wurde nach folgender Systematik vorgegangen: Zunächst wird zwischen Produktionsregeln für die Datenübertragung und für die Gefügesemantik unterschieden. Erstere dienen lediglich der Abbildung der Konstrukte aus der Prozessbeschreibungssprache auf das Produktbibliotheksmodell. So werden hier Produkte auf Ergebnistypen, Produktinhalte auf Inhaltstypen, und Erzeugende Abhängigkeiten auf Planungsvorgaben abgebildet. Die zweite Art von Produktionsregeln gliedert sich wiederum wie folgt in vier Unterarten:

- **Eröffnung:** Mit der Eröffnung wird das Grundgerüst der virtualisierten Produktionsregel erstellt. Sie kann bereits Elemente enthalten. In erster Linie sind dies solche, die immer vorhanden sein müssen. Beispielsweise ist der Musterknoten zur Erzeugung einer Soll-Version für jede erzeugende Abhängigkeit notwendig. Dies gilt genauso für den Verursacher, welcher hier durch den Suchmusterknoten mit der Id btr (*Betr*achteter) abgebildet wird. Dieser spannt hier gleichzeitig den Raum der möglichen Redexmorphismen auf, da dessen Belegung in die Transformationshistorie aufzunehmen ist[44]. Zusätzlich können auch Constraints und als Constraints wirkende Suchmusterknoten (Nutzung des Id-Patternmatchings oder der Ignorierung) angelegt werden.
- **Quantifizierung:** Jedes hierarchische Gefüge wird als ein Quantor (vgl. Prädikatenlogik) angesehen, zu dem ein entsprechender Suchmusterknoten erstellt wird.
- **Einschränkung:** Nicht-hierarchische Gefüge wirken als Prädikate zur Einschränkung möglicher Belegungen der Quantoren verwendet. Um die neu erstellten, virtualisierten Elemente der Produktionsregel an die gewünschten Quantoren zu binden, eignen sich die in Abbildung 5.47 eingeführten Notationen.

[44] Zwar ist auch die Belegung des Suchmusterknotens für den Ergebnistyps in die Transformationshistorie aufzunehmen. Durch das Id-Patternmatching sind jedoch die möglichen Belegungen auf genau eine konkrete Id eingeschränkt, wodurch der Raum möglicher Redexmorphismen eingeschränkt, aber nicht vergrößert wird.

- **Abschluss:** Mit Quantifizierungen und Einschränkungen werden aus fachlicher Sicht meist nur Bedingungen formuliert. Dabei ist es oft nur entscheidend, *dass* die Bedingung erfüllt ist – und nicht *wodurch*. Anders formuliert sind die Belegungen der entsprechenden Suchmusterknoten nicht in die Transformationshistorie aufzunehmen. Entsprechend nutzen beide Produktionsregelarten extensiv die Ignorierung (siehe Kapitel 2.4.3). Sofern so definierte Bedingungen sich dennoch gezielt auf einzelne Musterknoten aus der Eröffnung beziehen sollen, muss eine entsprechende Kopplung geschaffen werden. Dies lässt sich mit abschliessenden Produktionsregeln erzielen, die dann greifen können, sobald die zu koppelnde Eröffnung und Bedingung im ausreichenden Umfang erzeugt wurden[45]: In diesem Beispiel wird der Suchmusterknoten des Betrachteten (mit der Id btr) durch einen Constraint an den Quantor gekoppelt, der durch Produktionsregeln für die Quantifizierung für das entsprechende Gefüge in der Gefügekette erzeugt wurde. Die Kopplung besteht darin, dass die Belegungen beider Suchmusterknoten gleich sein müssen. Dies ist gleichbedeutend zu der Vereinigung beider Suchmusterknoten.

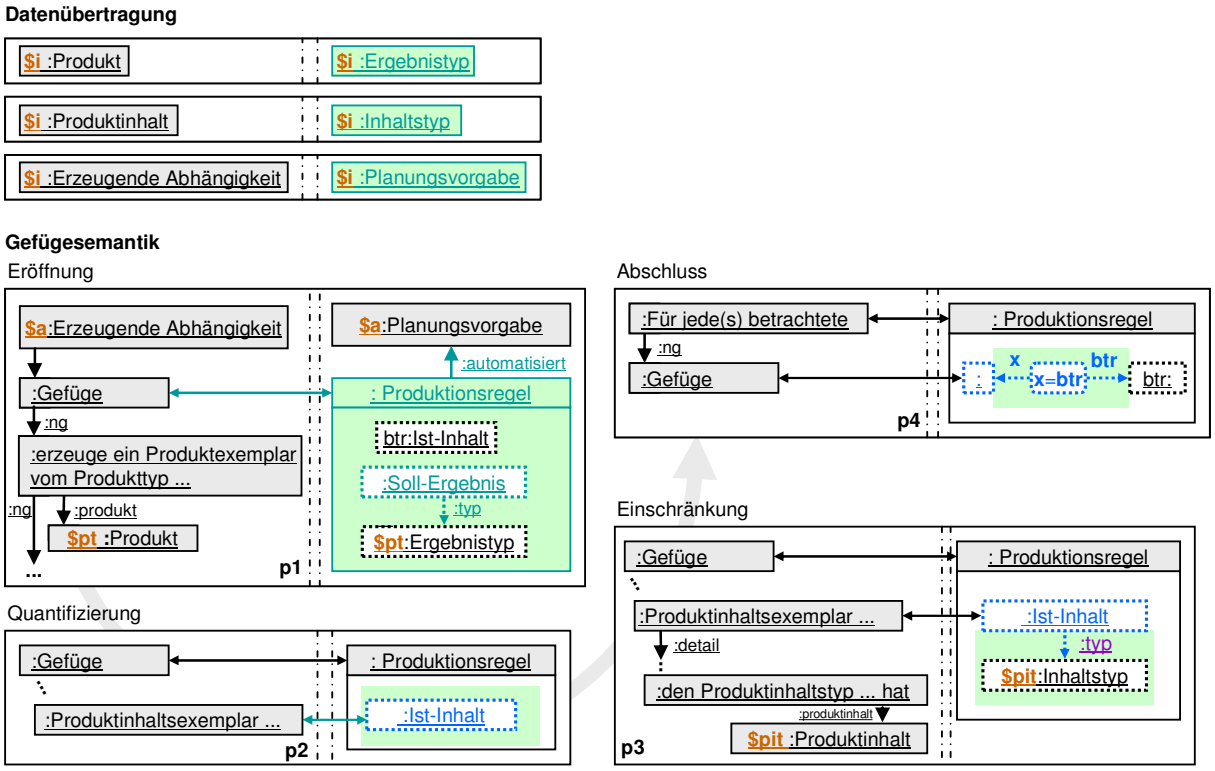


Abbildung 5.48 Beispiel einer Semantikabbildung

Die Anwendung der Produktionsregeln für die Gefügesemantik auf das laufende Beispiel ist in Abbildung 5.49 gezeigt. Als Ausgangssituation wird angenommen, dass die Produktionsregeln für die Datenübertragung bereits ausgeführt wurden. Auf dieser Basis werden die Produktionsregeln p1 – p4 der Reihenfolge nach ausgeführt.

[45] In diesem Fall könnte p4 bereits vor p3 greifen. Im Allgemeinen können aber in p4 auch die von p3 erzeugten Elemente als Voraussetzung definiert werden.

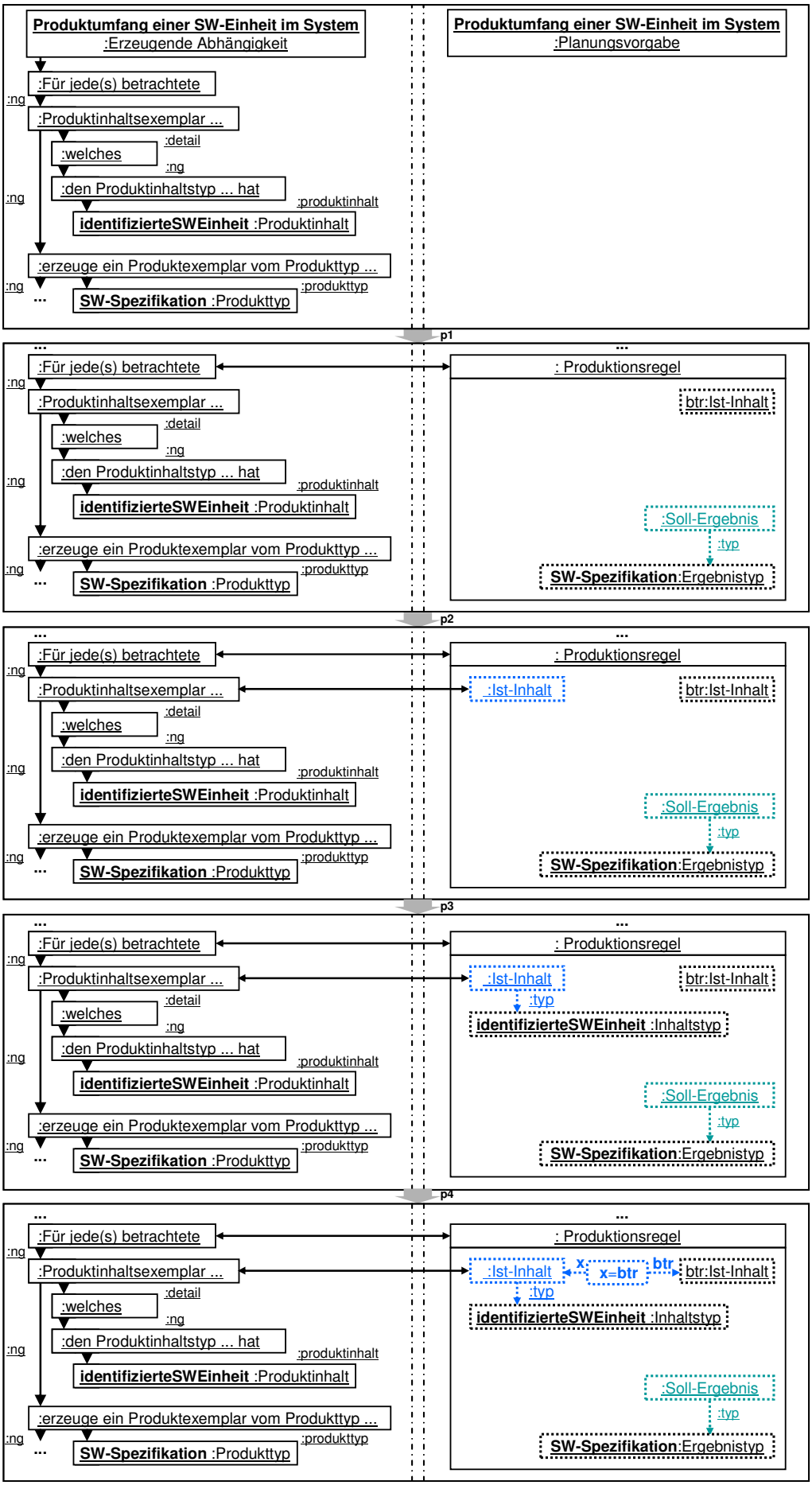


Abbildung 5.49 Wirkung der Produktionsregelarten für eine Gefügesemantik

5.5 Zusammenfassung

Der Lösungsansatz stellt bereits eine Zusammenfassung der Lösung dar, der wiederum in Kapitel 5.1.5 zusammengefasst ist. In diesem Kapitel werden daher der im Lösungsansatz noch nicht enthaltene Anteil – die konkrete Formalisierung – in kompakter Form präsentiert.

Abbildung 5.50 erweitert den Überblick der definierten Strukturen und diesbezüglich erweiternder Konzepte aus Abbildung 2.65 um Anteile, die durch die Grundlagen des Projektcontrollings sowie der Problemstellung und Lösung hinzukommenden. Entsprechend wird dieselbe Notation wie in Abbildung 2.65 verwendet.

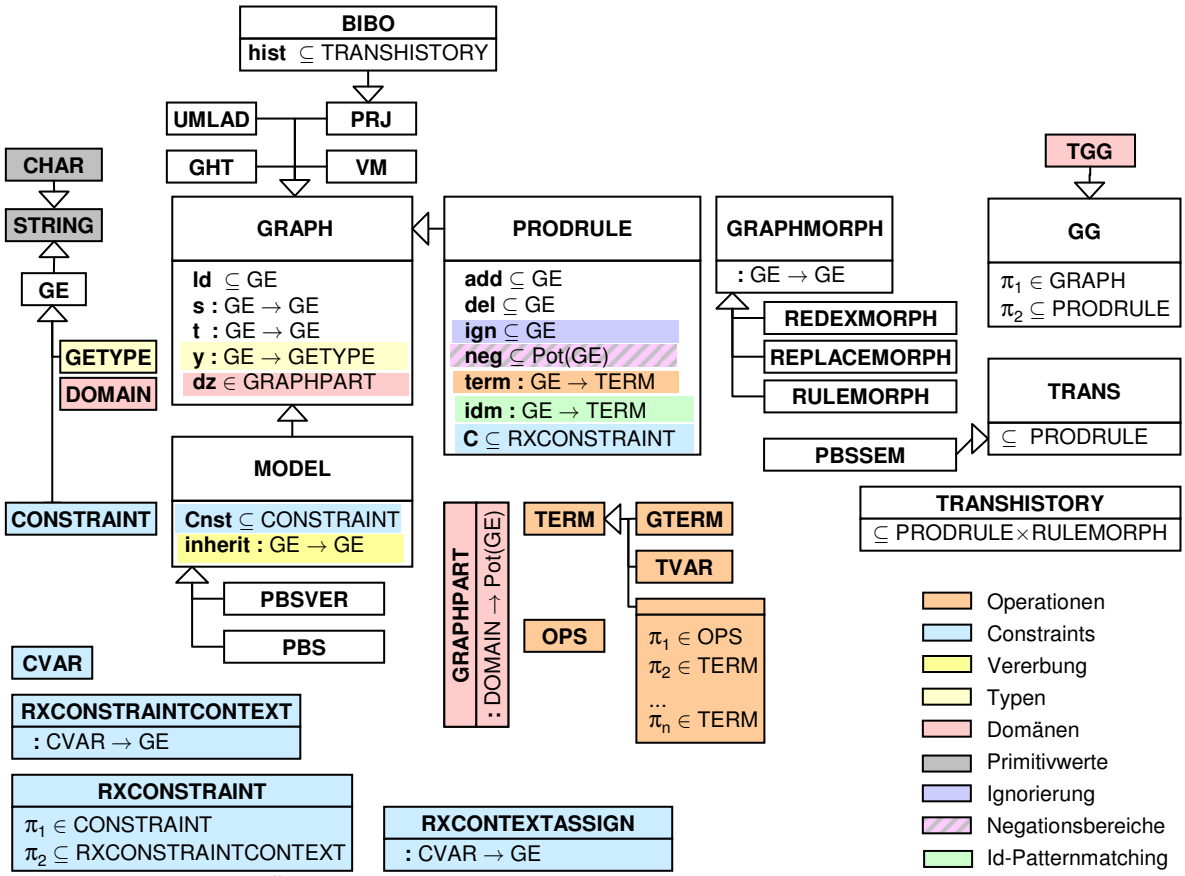
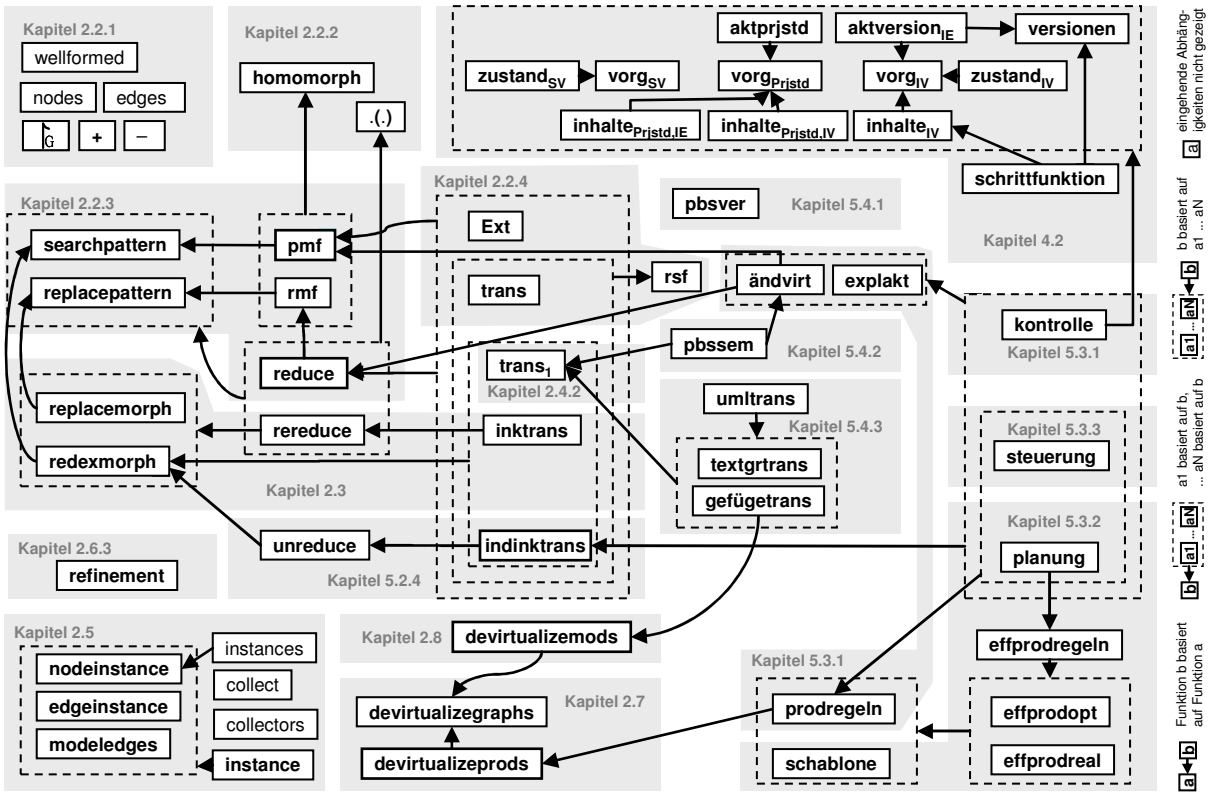


Abbildung 5.50 Überblick über definierte Strukturen und erweiternde Konzepte (gesamt)

Abbildung 5.51 zeigt die wesentlichsten, in dieser Arbeit definierten Funktionen und erweitert damit den Überblick aus Abbildung 2.67.



Kapitel 2.2.1 Graphen als Struktur für Wörter
$|_G$: GRAPH x Pot(GE) -> GRAPH
+ : GRAPH x GRAPH -> GRAPH
- : GRAPH x GRAPH -> GRAPH
wellformed : GRAPH -> BOOL
nodes : GRAPH -> Pot(GE)
edges : GRAPH -> Pot(GE)

Kapitel 2.2.2 Graphhomomorphismus
homomorph: GRAPHMORPH x GRAPH x GRAPH -> BOOL
.(.): GRAPHMORPH x GRAPH -> GRAPH

Kapitel 2.2.3 Produktionsregel
searchpattern: PRODRULE -> PRODRULE
replacepattern: PRODRULE -> PRODRULE
pmf: GRAPH x PRODRULE -> Pot(REDEXMORPH)
rmf: GRAPH x PRODRULE x REDEXMORPH -> REPLACEMORPH
reduce: GRAPH x PRODRULE x REDEXMORPH -> REPLACEMORPH

Kapitel 2.2.4 Graphgrammatik und Transformation
Ext: GG -> Pot(GRAPH)
rsf: Pot(PRODRULE x REDEXMORPH) -> PRODRULE x REDEXMORPH
trans: GRAPH x TRANS -> GRAPH

Kapitel 2.3 Aktualisierung einer Transformationsanwendung
replacemorph: RULEMORPH x PRODRULE -> REPLACEMORPH
redexmorph: RULEMORPH x PRODRULE -> REDEXMORPH
rereduce: GRAPH x PRODRULE x REDEXMORPH -> REPLACEMORPH
inktrans: GRAPH x TRANS x TRANSHISTORY-> GRAPH x TRANSHISTORY

Kapitel 2.4.2 Transformation mit Graphgrammatiken
trans$_1$: GRAPH x TRANS -> GRAPH

Kapitel 2.5 Modelle
nodeinstance: GE x GRAPH x GE x MODEL -> BOOL
edgeinstance: GE x GRAPH x GE x MODEL -> BOOL
modeledges: GE x MODEL -> Pot(GE)
instance: GRAPH x MODEL -> BOOL
instances: GETYPE x GRAPH [x MODEL] -> Pot(GE)
collect: GE x GETYPE x GRAPH [x MODEL] -> Pot(GE)
collectors: GE x GETYPE x GRAPH [x MODEL] -> Pot(GE)

Kapitel 2.6.3 Vererbung
refinement: MODEL x MODEL -> BOOL

Kapitel 2.7 Dynamische Transformation
devirtualizegraphs: GRAPH -> Pot(GRAPH)
devirtualizeprods: GRAPH -> Pot(PRODRULE)

Kapitel 2.8 Metamodelle
devirtualizemods: GRAPH -> Pot(MODEL)

Kapitel 4.2 Problemdomäne
vorg$_{IV}$: BIBO -> { f : GE -> GE }
vorg$_{SV}$: BIBO -> { f : GE -> GE }
vorg$_{Prjstd}$: BIBO -> { f : GE -> GE }
inhalte$_{IV}$: GE x BIBO -> Pot(GE)
inhalte$_{Prjstd,IV}$: GE x BIBO -> Pot(GE)
inhalte$_{Prjstd,IE}$: GE x BIBO -> Pot(GE)
zustand$_{IV}$: GE x BIBO -> Pot(GE)
zustand$_{SV}$: GE x BIBO -> Pot(GE)
versionen: GE x BIBO -> Pot(GE)
aktversion$_{IE}$: GE x BIBO -> GE
aktprjstd: GE x BIBO -> GE
schrittfunktion: {f:BIBO->BIBO} x MODEL -> BOOL

Kapitel 5.2.4 Induktiv inkrementelle Transformation
unreduce: GRAPH x PRODRULE x RULEMORPH -> GRAPH
indinktrans: GRAPH x TRANS x TRANSHISTORY -> GRAPH x TRANSHISTORY

Kapitel 5.3.1 Erweiterte Produktbibliothek
prodregeln: BIBO x GE -> Pot(PRODRULE)
explakt: PRODRULE -> PRODRULE
ändvirt: BIBO x BIBO x GE -> BIBO

Kapitel 5.3.2 Automatisierte Planung
schablone: GE x BIBO -> PRODRULE
effprodreal: GE x BIBO -> PRODRULE
effprodopt: GE x GE x BIBO -> PRODRULE
effprods: BIBO -> Pot(PRODRULE)
planung: BIBO -> BIBO

Kapitel 5.3.3 Teilautomatisierte Kontrolle
kontrolle: BIBO -> BIBO

Kapitel 5.3.4 Automatisierte Steuerung
steuerung: BIBO -> BIBO

Kapitel 5.4.1 Verfeinerungsabbildung
pbsver: MODEL -> MODEL

Kapitel 5.4.2 Semantikabbildung
pbssem: TRANS x GRAPH -> BIBO

Kapitel 5.4.3 Intuitive Formalisierung
textgrtrans: UMLAD -> GHT
gefügetrans: GHT -> PBS
umltrans: UMLAD -> PBS

Abbildung 5.51 Überblick über die Definitionen der Lösung

6 Evaluation

In diesem Kapitel wird die Lösung hinsichtlich ihrer Praxistauglichkeit evaluiert. Wie Abbildung 6.1 zeigt, wird dazu aus der Praxis das Fallbeispiel des Vorgehensmodells VM_0 mit der dazugehörigen Prozessbeschreibungssprache PBS_0 verwendet, die bereits in den Kapiteln 3.2.4 und 3.3 vorgestellt wurden. Kapitel 6.1 zeigt, wie PBS_0 unter Verwendung der Integrationsmethodik mit dem Automatisierungskonzept integriert wird, woraus PBS^+ entsteht. Kapitel 6.2 zeigt, wie VM_0 von PBS_0 nach PBS^+ migriert wird, worduch VM^+ entsteht. Die formale Semantik von VM^+ wird in Kapitel 6.3 gezeigt. Abschließend wird in Kapitel 6.4 darauf eingegangen, ob und wie sich die Lösung mit den in der Praxis verwendeten Werkzeugen integrieren lässt.

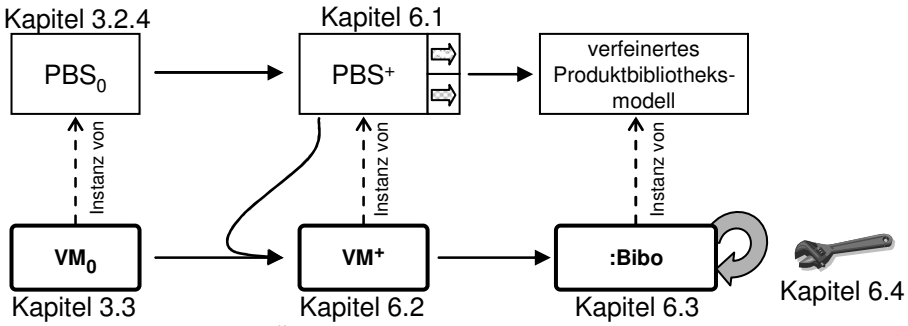


Abbildung 6.1 Überblick zum Aufbau des Evaluationskapitels

6.1 Mit dem Automatisierungskonzept integriertes PBS_0: PBS^+

In diesem Kapitel wird die Prozessbeschreibungssprache PBS^+ vorgestellt, die aus der Integration der Prozessbeschreibungssprache PBS_0 mit Automatisierungskonzept hervorgeht – konkret durch Anwenden des Schritts PBS-Integration aus der Integrationsmethodik. Kapitel 6.3.1 stellt die Syntax, Kapitel 6.3.2 die zugehörige Verfeinerungsabbildung und Kapitel 6.3.3 die darauf aufsetzende Semantikabbildung vor.

6.1.1 Syntax

Abbildung 6.2 zeigt im oberen Teil die grundlegenden Konstrukte von PBS^+, die weitestgehend aus PBS_0 übernommenen sind. Die dabei vorgenommenen Anpassungen umfassen folgende Punkte:

- Die Konstrukte Tailoringabhängigkeit(Von/Zu)Produkt und ErzeugendeAbhängigkeit (Von/Zu)Produkt wurden entfernt und durch eine Gefügekette ersetzt.
- Das Konstrukt Produktinhalt (sowie dessen Spezialisierungen) wurde(n) eingeführt um Vorgaben auch über Dokumentinhalten formalisieren zu können.
- Die Konstrukte des Zustandsmodells wurde aus der Produktbibliothek in PBS^+ kopiert, um dem Prozessingenieur die Möglichkeit zu geben, auch andere Zustandsmodelle als das eine des V-Modells zu definieren. Dementsprechend ist das Konstrukt des Zustandsübergangs ebenfalls mit einer Gefügekette verbunden.

Der untere Teil der zeigt die Terminale, aus denen Gefügeketten gebildet werden können. Diese sind thematisch sortiert: Beginnend mit den Einstiegspunkten für die drei Use Cases Erzeugende Abhängigkeit, Tailoringabhängigkeit und Zustandsübergang, werden anschließend die jeweils betrachteten Gegenstände wie Soll-Versionen, Produkt- oder Produktinhaltsexemplare[46] mit den jeweils diesbezüglich möglichen Einschränkungen gelistet.

[46] Ist-Ergebnisse werden im V-Modell XT als Produktexemplare. Produktinhaltsexemplare sei der dazu passende Begriff für Ist-Inhalte.

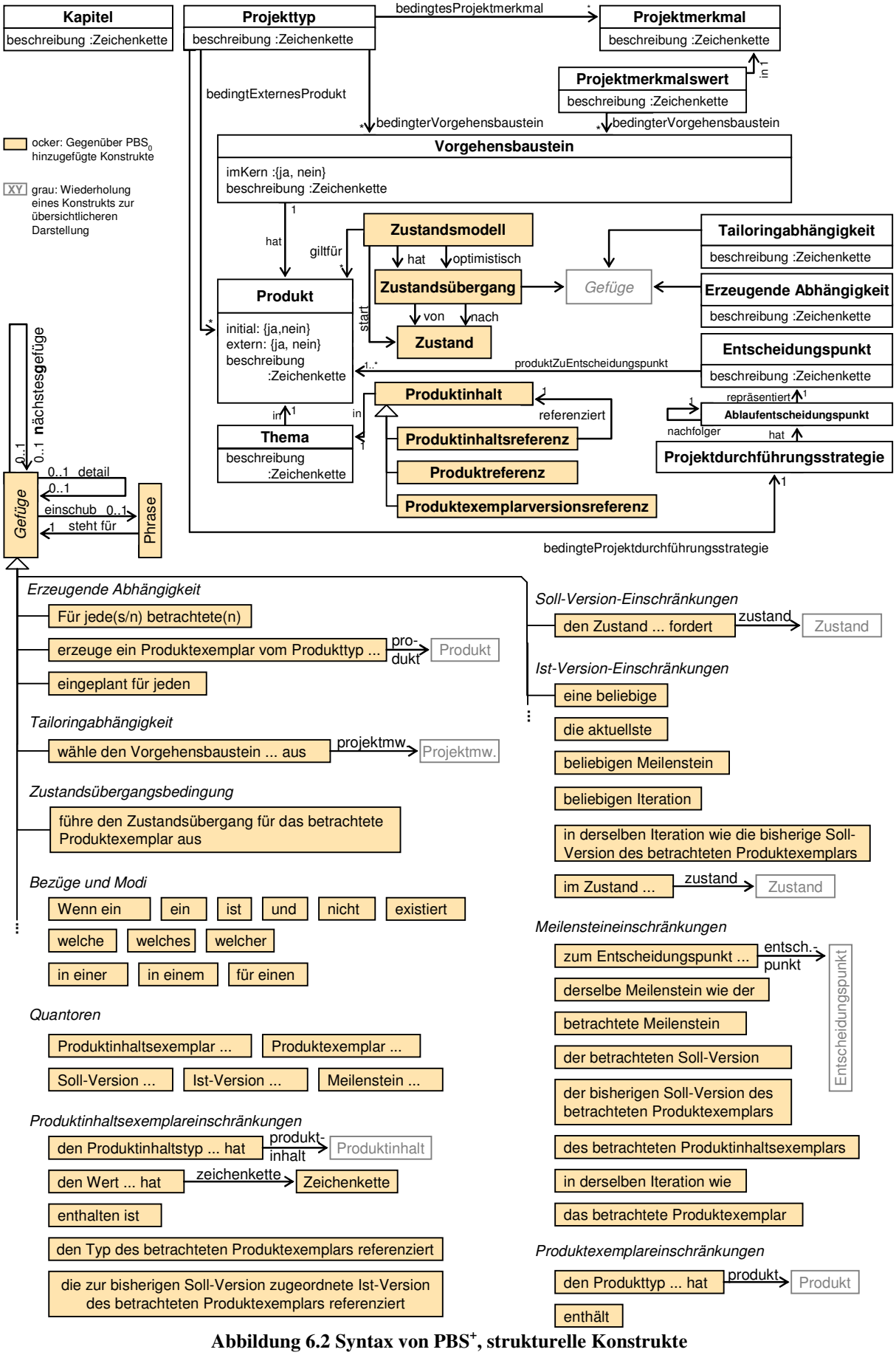


Abbildung 6.2 Syntax von PBS+, strukturelle Konstrukte

Auf welche Weise diese Terminale zu Gefügeketten verknüpft werden können, wird durch die UML-Aktivitätsdiagramme in den restlichen Abbildungen dieses Kapitels vorgestellt. Diese UML-Aktivitätsdiagramme sind das Ergebnis des Schritts Entwurf in der intuitiven Formalisierung. Sie werden hier anstelle der daraus abzuleitenden Constraints gezeigt, da, obwohl Teil der PBS$^+$-Syntax, sie nicht zum „Lesen“ sondern nur für eine werkzeuggestützten Wortprüfung ausgelegt sind. Gleichwohl eignen sich die Constraints um bei einem Fehler aussagekräftige Fehlermeldungen abzuleiten da bei einer fehlgeschlagenen Constraintprüfung nicht nur genau die betroffene Stelle der geprüften Gefügekette vorliegt, sondern auch das, was als nächstes erwartet wurde.

Das UML-Aktivitätsdiagramm in Abbildung 6.3 stellt den Startpunkt für eine formale Definition einer erzeugenden Produktabhängigkeit dar. Der Idee des Automatisierungskonzeptes folgend, wird ein Element aus der Produktbibliothek als das „betrachtete“ dargestellt. Dadurch kann in den weiteren Terminalen allein über deren Benennung ein Bezug hergestellt werden – ohne Variablen einführen zu müssen, die dann weitere Assoziationen erfordern würden. Der Terminus der Erzeugung eines Produktexemplars ist an der Begrifflichkeit des V-Modells ausgerichtet – im Bezug auf das Automatisierungskonzept ist die Erzeugung eines Soll-Ergebnisses zu verstehen.

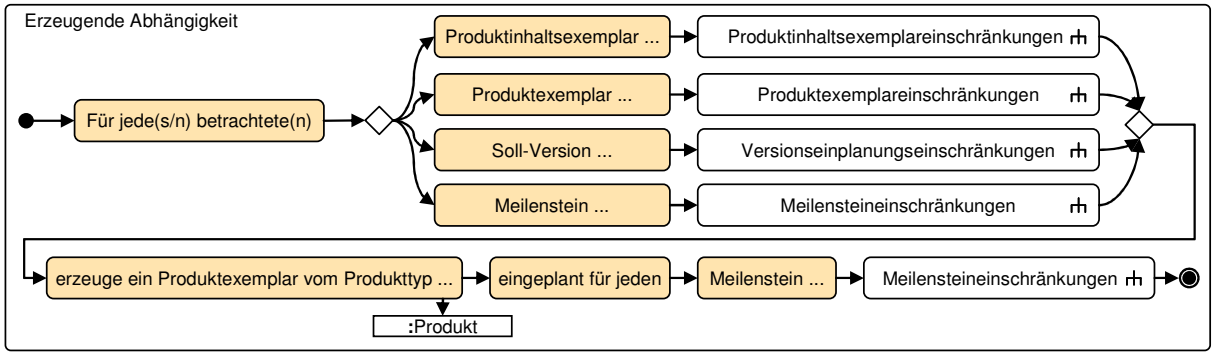


Abbildung 6.3 Schema für Erzeugende Abhängigkeiten

Bei Tailoringabhängigkeiten handelt es sich um das dynamische, also auf den bisherig im Projekt erarbeiteten Ergebnissen basierende Steuerungsvorgaben. Dementsprechend werden Gefügeketten mit der Existenz eines bestimmten Produktinhaltsexemplars eingeleitet und dessen Folge ein Projektmerkmalswert als auszuwählen definiert (Abbildung 6.4).

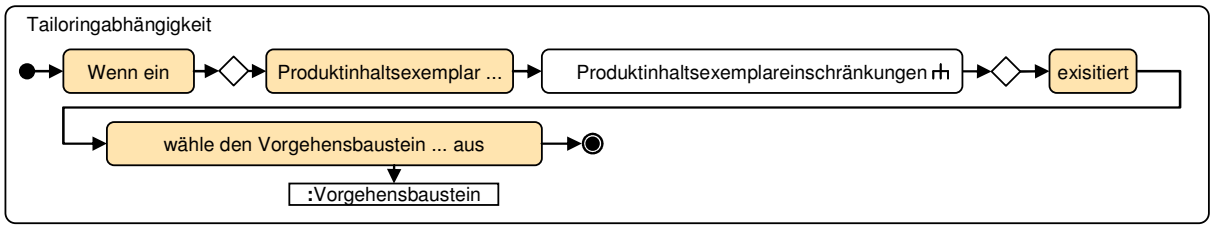


Abbildung 6.4 Schema für Tailoringabhängigkeiten

Bei Zustandsübergangsbeziehungen (Abbildung 6.5) ist das Betrachtete das Produktexemplar, für welches die Durchführbarkeit des Zustandsübergangs geprüft wird. Ansonsten handelt es sich auch hier – wie bei Tailoringabhängigkeiten – um Bedingungen die auf ähnliche Weise konstruiert werden.

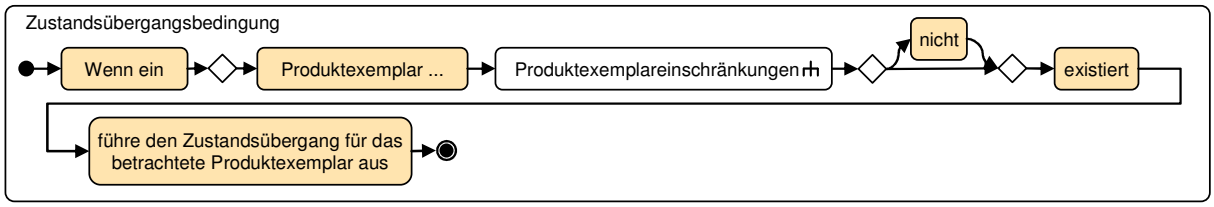


Abbildung 6.5 Schema für Zustandsübergangsbedingungen

Abbildung 6.6 zeigt die Möglichkeiten zur Einschränkung von Produktinhaltsexemplaren. Das erste Terminal (welches) dient dabei lediglich als Füllwort um den Lesefluss der entstehenden Gefügeketten zu verbessern. Ein weiteres solches Mittel ist die Verwendung von „…“ als Teil eines Terminals – insbesondere auch in dessen Inneren. Durch Verwendung von Hierarchien wird der entsprechende Teil ohnehin strukturell als Detail abgelegt und kann werkzeugtechnisch ausgeblendet werden. Bei andernfalls gezwungener Umstellung der Wörter des Terminals vor den variablen Teil, wird im Allgemeinen das intuitive Verständnis erschwert. Dieser Effekt kann beispielsweise in Programmiersprachen bei Funktionsnamen beobachtet werden, bei denen, da sämtliche Parameter nach dem „Terminal“ des Funktionsnamens stehen, die Auswirkung der Parameter auf das Verhalten der Funktion nicht immer sofort ersichtlich ist. Darüber hinaus zeigen die Konstrukte aus Abbildung 6.6, dass nicht um jeden Preis versucht werden muss, komplexe Bedingungen durch eine kleine Anzahl von orthogonalen Konstrukten darzustellen. Schließlich zeigt Abbildung 6.6 auch die Möglichkeit über das Konstrukt und Schleifen zu bilden, wodurch hier mehrere Einschränkungen kombiniert werden können.

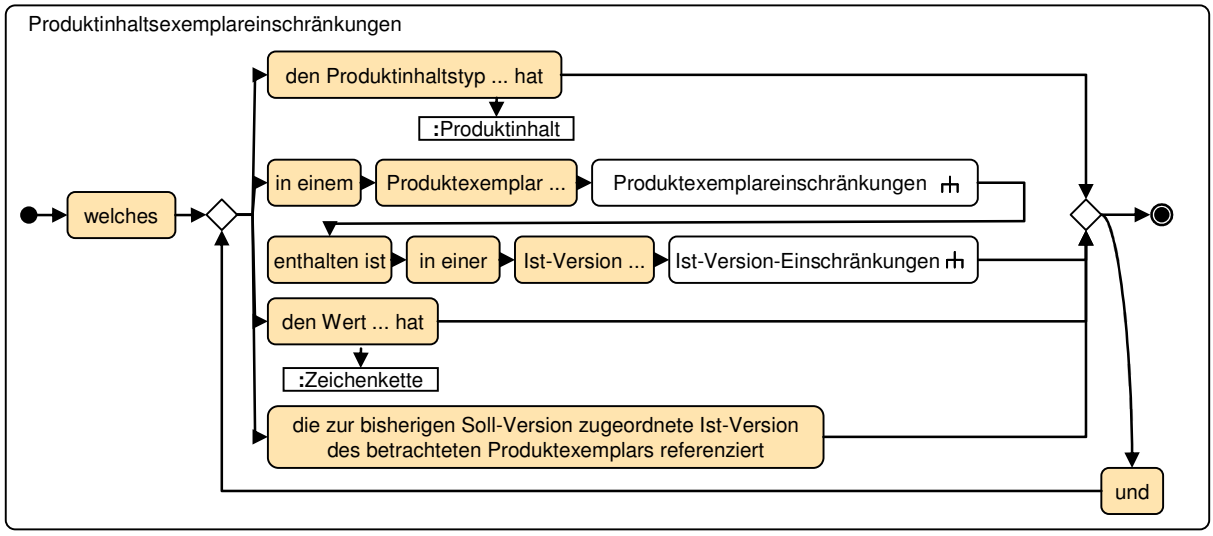


Abbildung 6.6 Schema für Produktinhaltsexemplareinschränkungen

Die Einschränkungen der Produktexemplare, in Abbildung 6.7, sind ähnlich wie die zur Einschränkung von Produktinhaltsexemplaren ausgelegt. Beide Einschränkungen zusammen zeigen, dass auch rekursive Sprachen definiert werden können. So kann eine Vorgabe definiert werden, bei der ein Produktinhaltsexemplar sich in einem bestimmten Produktexemplar befinden muss, welches wiederum ein (anderes) Produktinhaltsexemplar mit etwaigen Einschränkungen enthält. Darüber hinaus können bei einer Produktexemplar-einscrhänkung mehrere Produktinhaltsexemplareinschränkungen zur selben Ist-Version gestellt werden (innere und-Schleife).

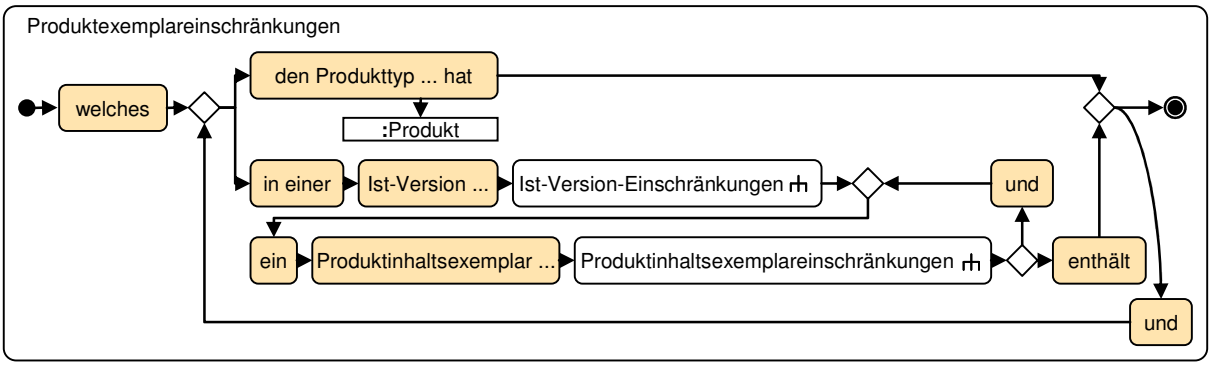


Abbildung 6.7 Schema für Produktexemplareinschränkungen

In Abbildung 6.8 wird das UML-Aktivitätsdiagramm für die Einschränkungen der zu betrachtenden Ist-Versionen vorgestellt. Wie später bei der Definition der Semantikabbildung in Kapitel 6.2.3 gezeigt, kann eine kontextfreie Grammatik auch mit einer kontextsensitiven Semantik auf einfache Weise unterlegt werden. Konkret betrifft dies die Kombinationen des Konstrukts die aktuellste ist mit einem der vier nachfolgenden Konstrukte.

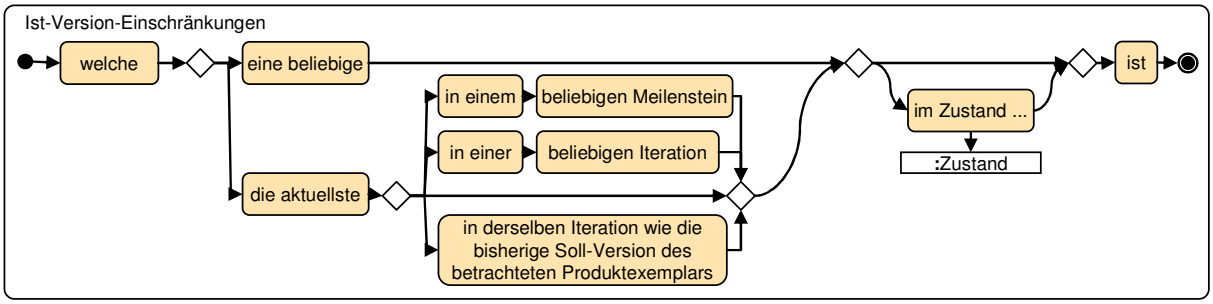


Abbildung 6.8 Schema für Ist-Version-Einschränkungen

Abbildung 6.9 und Abbildung 6.10 vervollständigen das UML-Aktivitätsdiagramm mit Konstrukten für die Einschränkung von Soll-Versionen und Meilensteinen.

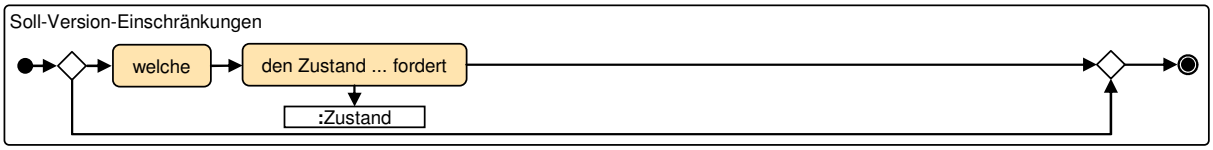


Abbildung 6.9 Schema für Soll-Version-Einschränkungen

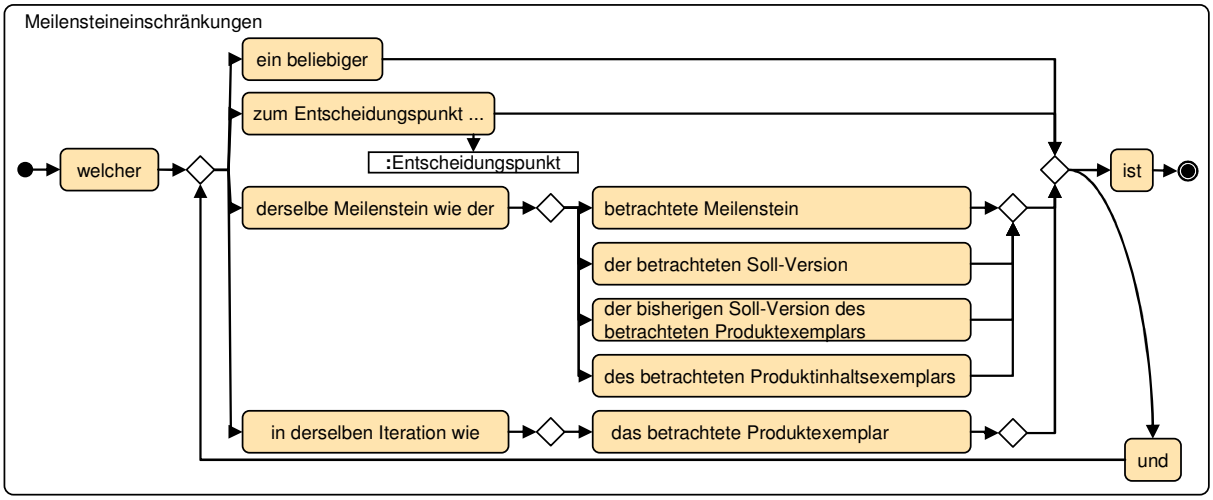


Abbildung 6.10 Schema für Meilensteineinschränkungen

6.1.2 Verfeinerungsabbildung

Wie in der Lösung im Kapitel 5.4.1 dargestellt, kann eine Verfeinerungsabbildung das Produktbibliotheksmodell prinzipiell hinsichtlich aller vier Teile des Projektcontrollings verfeinern. Die hier definierte Verfeinerungsabbildung (Abbildung 6.11) umfasst jedoch nur die Teile zu Strg, Soll und Ist.

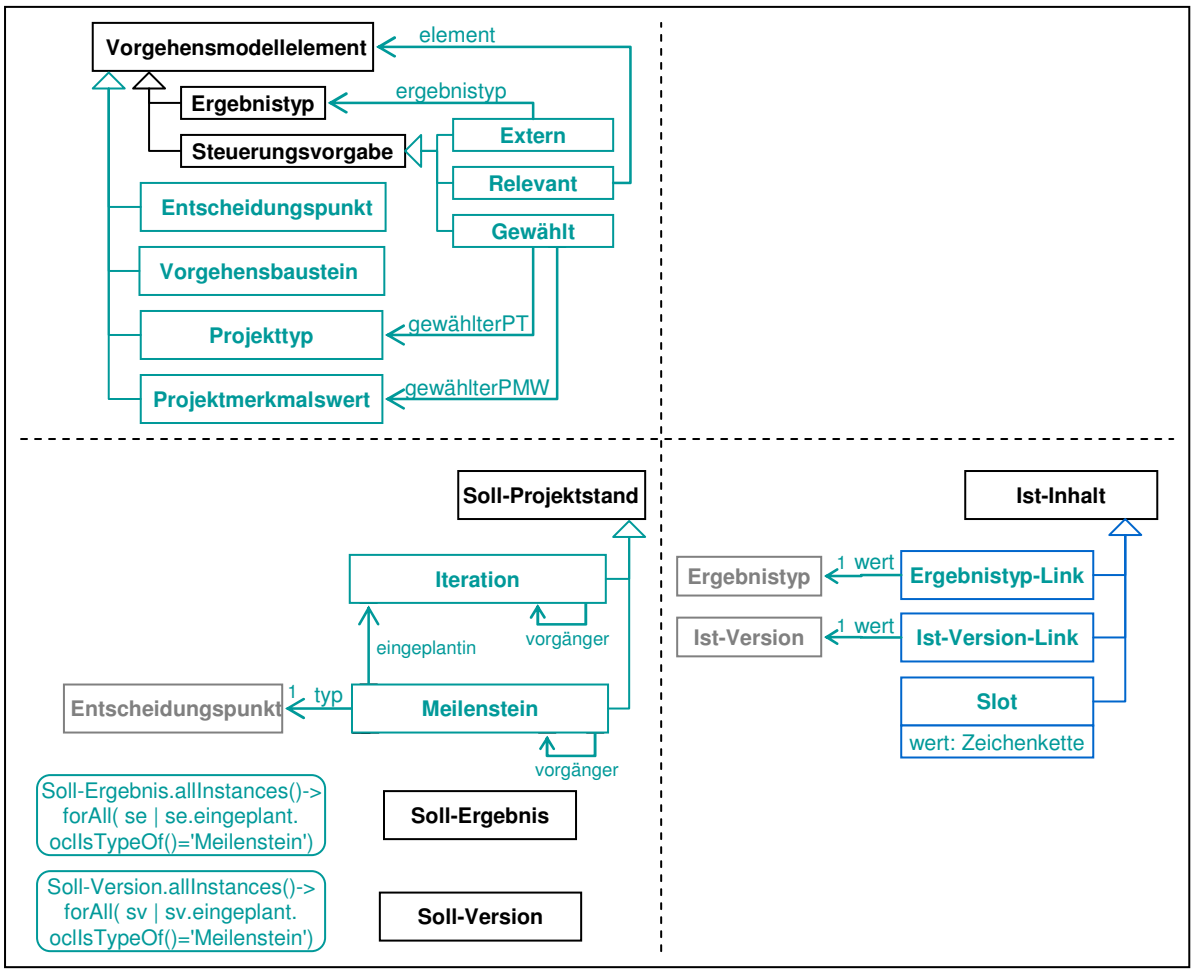


Abbildung 6.11 Verfeinerungsabbildung für PBS+

Zur Verfeinerung der Steuerung könnten prinzipiell alle Konstrukte aus PBS+ übernommen werden. Die hier vorgestellte Verfeinerungsabbildung verfolgt die Idee, nur solche zu übernehmen, die von zentraler Bedeutung sind. Daher entfallen zunächst all die Konstrukte von PBS+, für die es im Produktbibliotheksmodell bereits eine Entsprechung gibt. Konkret betrifft das die Konstrukte Produkt und Produktinhalt (abbildbar auf Ergebnistyp und Inhaltstyp), Erzeugende Abhängigkeit und Tailoringabhängigkeit (abbildbar auf Planungsvorgabe und Strukturvorgabe) sowie alle Konstrukte des Zustandsmodells. Weiterhin entfallen die Konstrukte Projektdurchführungsstrategie und Ablaufentscheidungspunkt, da diese nur als Constraints bei der manuellen Definition von Soll-Projektstand-Ketten (im V-Modell als Projektdurchführungspläne bezeichnet) wirken. Die Konstrukte Kapitel und Thema haben im Produktbibliotheksmodell keine Entsprechungen und entfallen entsprechend – deren informale Vorgaben werden jedoch auf andere Konstrukte aufgeteilt (beispielsweise werden Themen-Instanzen durch entsprechende Inhaltstyp-Instanzen abgebildet). Schließlich wird auch das Konstrukt Projektmerkmal nicht in die Verfeinerung des Produktbibliotheksmodells aufgenommen, da die Semantikabbildung an dieser Stelle auch über der Kombination aus Projekttyp und Projektmerkmalswerte definiert werden kann. Für die Verfeinerung verbleiben aus PBS+ die Konstrukte Projekttyp, Projektmerkmalswert, Vorgehensbaustein und

Entscheidungspunkt, wobei auf das jeweilige Attribut beschreibung verzichtet wird, da es weder vom Automatisierungskonzept noch von der Semantikabbildung verarbeitet werden kann. Stattdessen werden die Steuerungsvorgaben Relevant, Extern und Gewählt eingeführt. Diese können jeweils bestimmte Vorgehensmodellelemente referenzieren, und dienen damit als Gegenstand höherstufiger Steuerungsvorgaben.

Zur Verfeinerung der Planung werden Soll-Projektstände in Iterationen und Meilensteine unterschieden. Jedem Meilenstein wird dabei ein Entscheidungspunkt als Typ zugeordnet. Zusätzlich, in der Abbildung nicht dargestellt, sei zugesichert, dass pro Iteration und Entscheidungspunkt es maximal einen Meilenstein geben kann. Somit können Planungsvorgaben auf einfache Weise erzeugte Soll-Elemente dem Zeitplan eindeutig zuordnen.

Zur Verfeinerung der Durchführung werden spezielle Ist-Inhalte eingeführt, um in der Semantikabbildung auf Werte von Ist-Inhalt-Instanzen zugreifen zu können. Ein Ist-Inhalt-Link ist der Ist-Inhalt zu einer Produktinhaltsreferenz, ein Ergebnistyp-Link der zu einer Produktreferenz, und der Ist-Version-Link zu einer Produktexemplarversionsreferenz. Für einfache Produktinhalte wird dagegen ein Slot verwendet.

6.1.3 Semantikabbildung

Die in diesem Kapitel vorgestellte Semantikabbildung ordnet einem in PBS$^+$ beschriebenen Vorgehensmodell eine Semantik in Form einer initialen Produktbibliothek zu. Die Produktionsregeln der Semantikabbildung folgen der Implementierungs-Methodik der Intuitiven Formalisierung (siehe Kapitel 5.4.3.2). Zu den zwei darin beschriebenen Arten (Datenübertragung und Gefügesemantik) seien zwei zusätzliche definiert, wodurch folgendes Gesamtbild der eingesetzten Produktionsregel-Arten ergibt:

- **Datenübertragung:** Produktionsregeln für die Abbildung von Daten aus dem integrierten Vorgehensmodell auf die Teilbibliothek Strg.
- **Initialelemente:** Produktionsregeln für die Erzeugung initialer Elemente für die Teilbibliothek Strg.
- **Grundsemantik:** Produktionsregeln für die Abbildung von Konstrukten, die keine Gefügetypen sind, auf (virtualisierte) Produktionsregeln für die Teilbibliothek Strg.
- **Gefügesemantik:** Produktionsregeln für die Abbildung von Gefügetypen auf (virtualisierte) Produktionsregeln für die Teilbibliothek Strg.

Abbildung 6.12 zeigt die Produktionsregeln zur Datenübertragung. Wie im Beispiel aus der Implementierungs-Methodik (Abbildung 5.48) werden auf Ergebnistypen, Produktinhalte auf Inhaltstypen und erzeugende Abhängigkeiten auf Planungsvorgaben abgebildet. Nach gleicher Systematik werden auch Tailoringabhängigkeiten auf Strukturvorgaben abgebildet. Zustandsübergänge werden auf Zustandsübergänge abgebildet, da für PBS$^+$ dieselben Konstrukte aus dem Produktbibliotheksmodell verwendet wurden (siehe Kapitel 6.2.1). Entsprechend der Verfeinerungsabbildung werden auch Entscheidungspunkte, Projekttypen und –merkmalswerte übertragen.

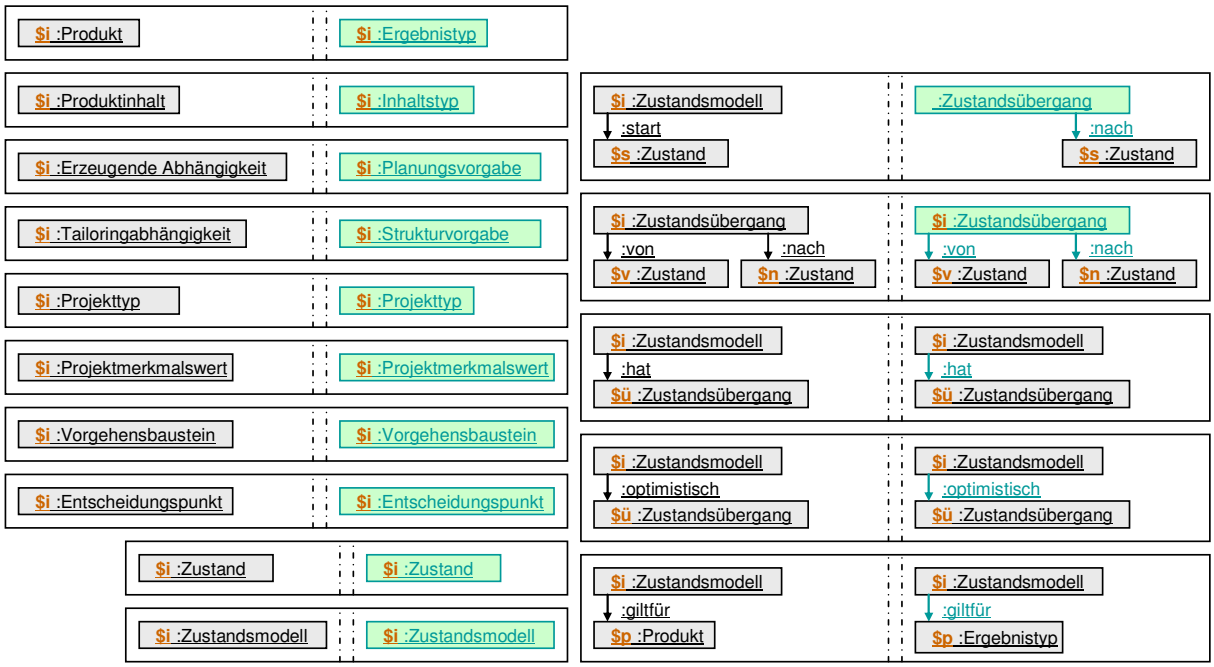


Abbildung 6.12 Semantikabbildung: Produktionsregeln für Datenübertragung

Abbildung 6.13 zeigt die Produktionsregeln für die Initialelemente. Diese greifen das Beispiel aus der Problemstellung (Abbildung 4.2) auf. So gibt es eine Steuerungsvorgabe für das statische Tailoring, eine für das Dynamische und eine für die Initialplanung. Diese Steuerungsvorgaben dienen lediglich als Anker um diesen die virtualisierten Produktionsregeln zuordnen zu können.

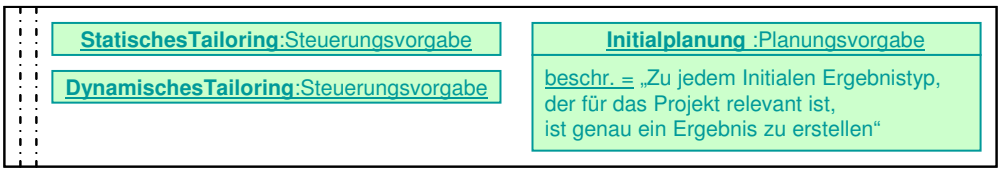


Abbildung 6.13 Semantikabbildung: Produktionsregeln für Initialelemente

Abbildung 6.14 zeigt die Produktionsregeln für die Grundsemantik des Tailorings. Die Produktionsregeln tp1 und tp2 erzeugen die virtualisierten Produktionsregeln zur Automatisierung des statischen Tailorings. Gemäß den Organisationstufen aus Abbildung 6.13 wird der Projekttyp vor Beginn eines Projektes gewählt und fixiert. Vorgehensbausteine sind zwar im Allgemeinen auch während eines Projekts hinzunehmbar – die im Kern liegenden bilden jedoch eine Ausnahme: Sie müssen immer vorhanden sein. Die Produktionsregeln tp3 bis tp6 erzeugen die virtualisierten Produktionsregeln für die Automatisierung des dynamischen Tailorings. in Abhängigkeit des gewählten Projekttyps und der gewählten Projektmerkmalswerte werden die Vorgehensbausteine und Ergebnistypen als relevant bzw. extern gesetzt.

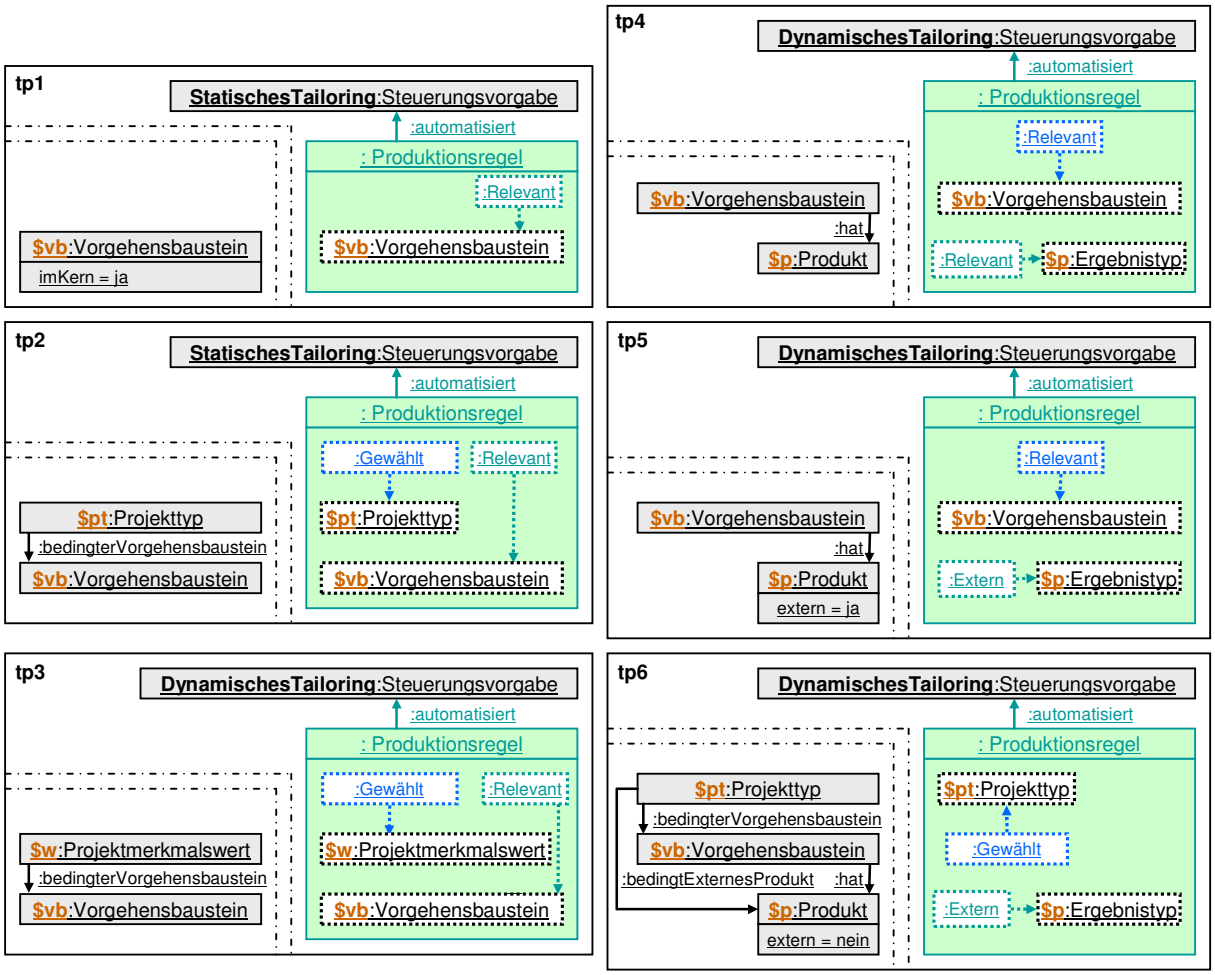


Abbildung 6.14 Produktionsregeln für die Grundsemantik (1/2)

Abbildung 6.15 behandelt die Grundsemantik initialer Produkttypen sowie die Zuordnung von Produkten zu Entscheidungspunkten. Für jedes initiale Produkt wird durch ip1 für die Produktbibliothek eine virtualisierte Produktionsregel erstellt, die genau für diesen Ergebnistyp ein Soll-Ergebnis allgemein einplant. Mit ip2 wird diese Einplanung konkret zu jedem Meilenstein eingeplant, zu dessen Entscheidungspunkt der Ergebnistyp zugeordnet ist. Durch ip3 wird eine virtualisierte Produktionsregel erstellt, die zu jedem nicht-initialen Produkttyp pro Meilenstein ein neues Soll-Ergebnis einplant.

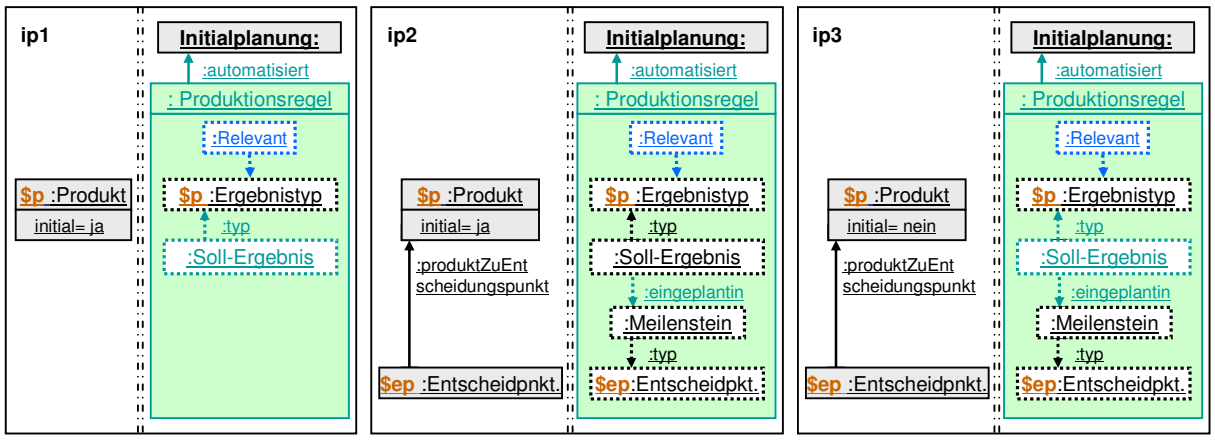


Abbildung 6.15 Produktionsregeln für die Grundsemantik (2/2)

Im restlichen Teil dieses Kapitels werden die Produktionsregeln für die Gefügesemantik behandelt. Abbildung 6.16 zeigt die Produktionsregeln für die Eröffnung von virtualisierten

Produktionsregeln zu erzeugenden Abhängigkeiten, Tailoringabhängigkeiten und Zustandsübergängen. Mit Abbildung 6.17 folgen die Produktionsregeln für Quantifizierungen. Abbildung 6.18 bis Abbildung 6.22 enthalten die Produktionsregeln für Einschränkungen und Abbildung 6.23 schließlich die Produktionsregeln für den Abschluss.

Gegenüber dem Beispiel aus der Implementierungs-Methodik (Abbildung 5.48) wird mit ea1 (Abbildung 6.16) in der Eröffnung einer virtualisierten Produktionsregel für erzeugende Abhängigkeiten der Raum möglicher Redexmorphismen neben dem (als Verursacher) betrachteten Ist-Inhalt, zusätzlich durch einen Meilenstein aufgespannt, zu dem das erzeugte Soll-Ergebnis eingeplant wird. Durch die Steuerungsvorgabe mit der Id relevant wird die Ausführbarkeit der Regel darauf beschränkt, dass der Ergebnistyp des einzuplanenden Soll-Ergebnisses für das Projekt auch relevant ist.

Mit ta1 wird eine virtualisierte Produktionsregel für eine Tailoringabhängigkeit eröffnet. Der Raum möglicher Redexmorphismen ist auf einen Redexmorphismus beschränkt, da die mögliche Belegung beider Suchmusterknoten durch das Id-Patternmatching eindeutig bestimmt ist. Erzeugt wird eine Kante, die beide Suchmusterknoten verbindet, und somit den in der Gefügekette referenzierten Vorgehensbaustein automatisch als gewählt setzt. Erst durch weitere Suchmusterknoten – die durch Quantifizierungen und Einschränkungen erzeugt werden – wird die Ausführbarkeit der virtualisierten Produktionsregel eingeschränkt. Dabei ist darauf zu achten, dass keine weiteren Suchmusterknoten hinzukommen, die den Raum möglicher Redexmorphismen erhöhen. Da die Produktionsregeln für Quantifizierungen und Einschränkungen auch für andere Vorgabearten wiederverwendet werden, erzeugen diese generell nur Suchmusterknoten die für die Transformationshistorie zu Ignorieren sind (bzw. solche, deren Belegung durch das Id-Patternmatching eindeutig bestimmt ist).

Mit zü1 wird eine virtualisierte Produktionsregel für eine Zustandsübergangsbedingung eröffnet. Diese enthält zwei (virtualisierte) Suchmusterknoten die (nicht-virtualisiert) in den Schablonen für die Versionsplanungen vorkommen (siehe Abbildung 5.28). Damit können die Zustandsübergangbedingungen auf die Belegungen der Schablone zugreifen, da diese Suchmusterknoten bei der Graphaddition die Verbindungsstelle bilden.

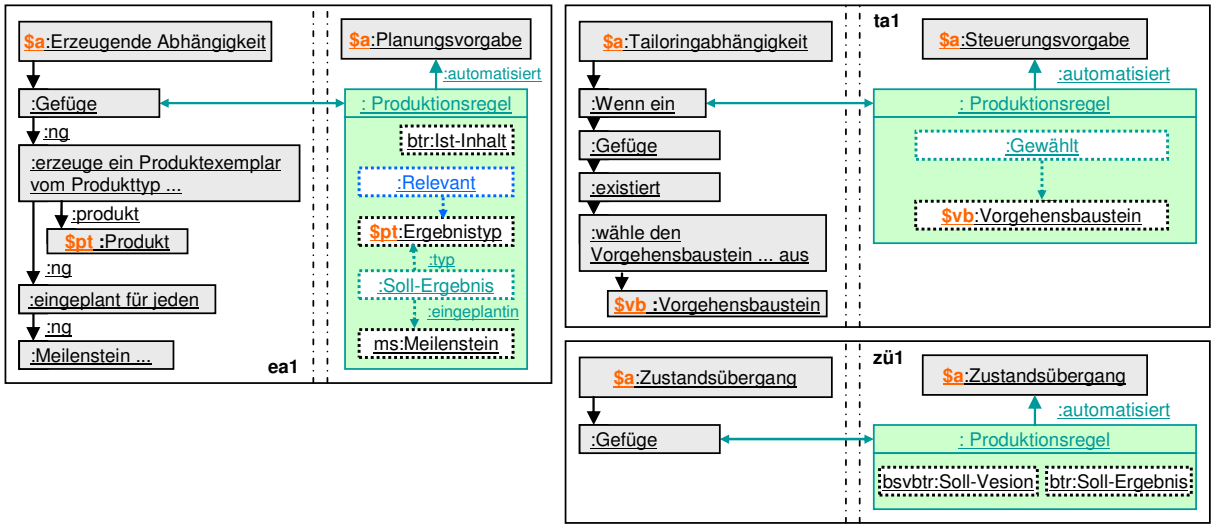


Abbildung 6.16 Produktionsregeln zur Eröffnung

Die Produktionsregeln für die Quantifizierungen (Abbildung 6.17) erzeugen einen Suchmusterknoten mit zum jeweiligen Gefügekonstrukt passenden Typ. Gegenüber dem Beispiel aus der Implementierungs-Methodik sind hier nicht nur Ist-Inhalte, sondern auch Ist-Ergebnisse, Versionen und Meilensteine quantifizierbar.

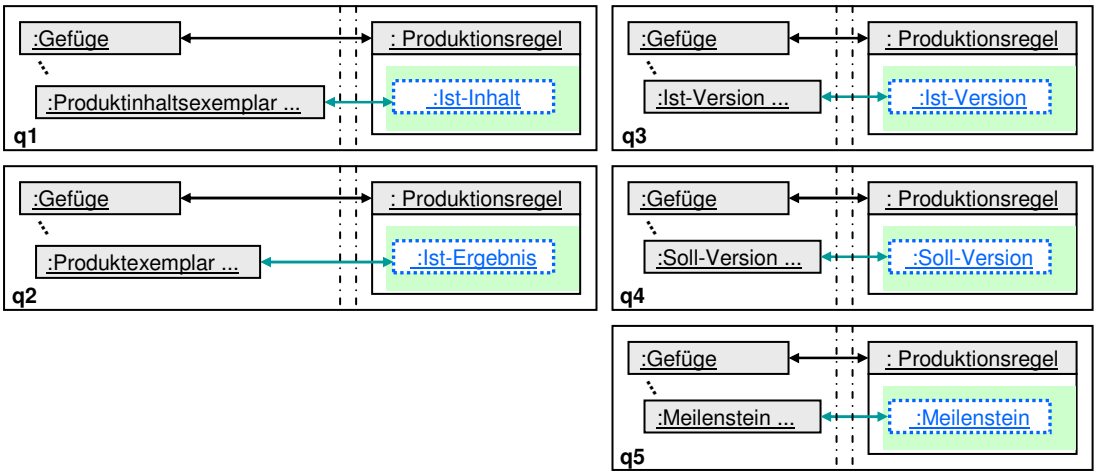


Abbildung 6.17 Produktionsregeln zur Quantifizierung

Abbildung 6.18 zeigt die Produktionsregeln für die Semantik von Produktexemplar-einschränkungen. Die Einschränkung anhand des Typs (Produktionsregel pxe2) ist analog zum dem Beispiel aus der Implementierungs-Methodik (Abbildung 5.48, Produktionsregel p3). Mit der Produktionsregel pxe1 werden dagegen mehrere Quantoren untereinander verbunden – was in der Prädikatenlogik der Verwendung eines mehrstelligen Prädikäts mit unterschiedlichen, gebundenen Variablen entspricht.

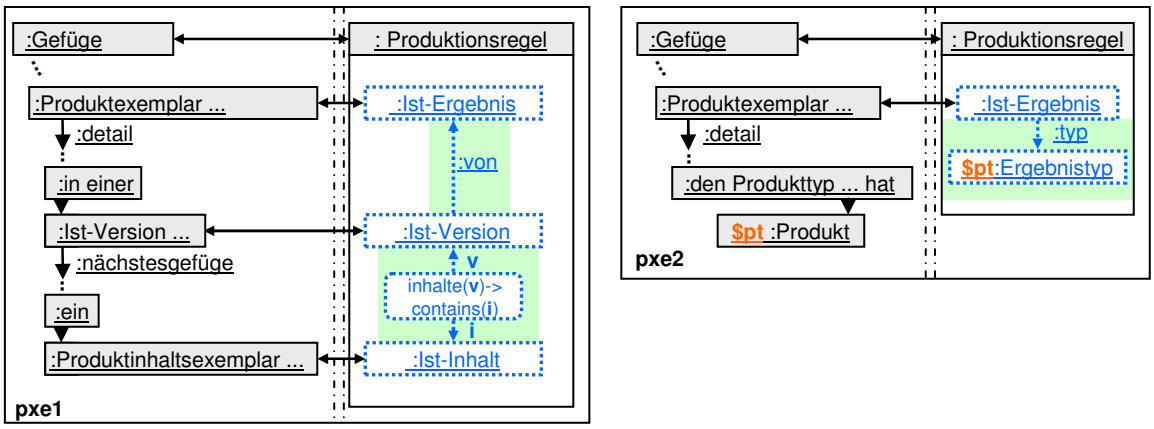


Abbildung 6.18 Produktionsregeln für Produktexemplareinschränkungen

Abbildung 6.19 zeigt die Produktionsregeln für die Semantik von Produktinhaltsexemplar-einschränkungen. Die Produktionsregeln pixe1 und pixe2 sind analog zu pix1 und pix2 für Produktexemplareinschränkungen. Durch pixe3 wird die Quantifizierung des übergeordneten Ist-Inhalts auf solche beschränkt, die ein Slot sind (vgl. Verfeinerungsabbildung in Abbildung 6.11), und den in der Gefügekette angegebenen Wert haben. Die Einschränkung erfolgt hier durch Erzeugung eines neuen Suchmusterknotens (mit Typ Slot), der über ein Constraint an den bisherigen gekoppelt wird. Diese Technik wird auch in den letzten beiden Produktionsregeln fortgesetzt: Bei pixe4 ist ein Ergebnistyp-Link, bei pixe5 ein Ist-Versionen-Link als Ist-Inhalt gesucht. Zusätzlich wird diese Spezialisierungstechnik in pixe4 angewendet, um den Typ des Betrachteten (der Suchmusterknoten aus der Eröffnung) auf ein Ist-Ergebnis einzuschränken. In pixe5 gibt es ebenfalls Bezüge zu den Suchmusterknoten aus der Eröffnung – diesmal jedoch ohne diese Im Typ zu spezialisieren.

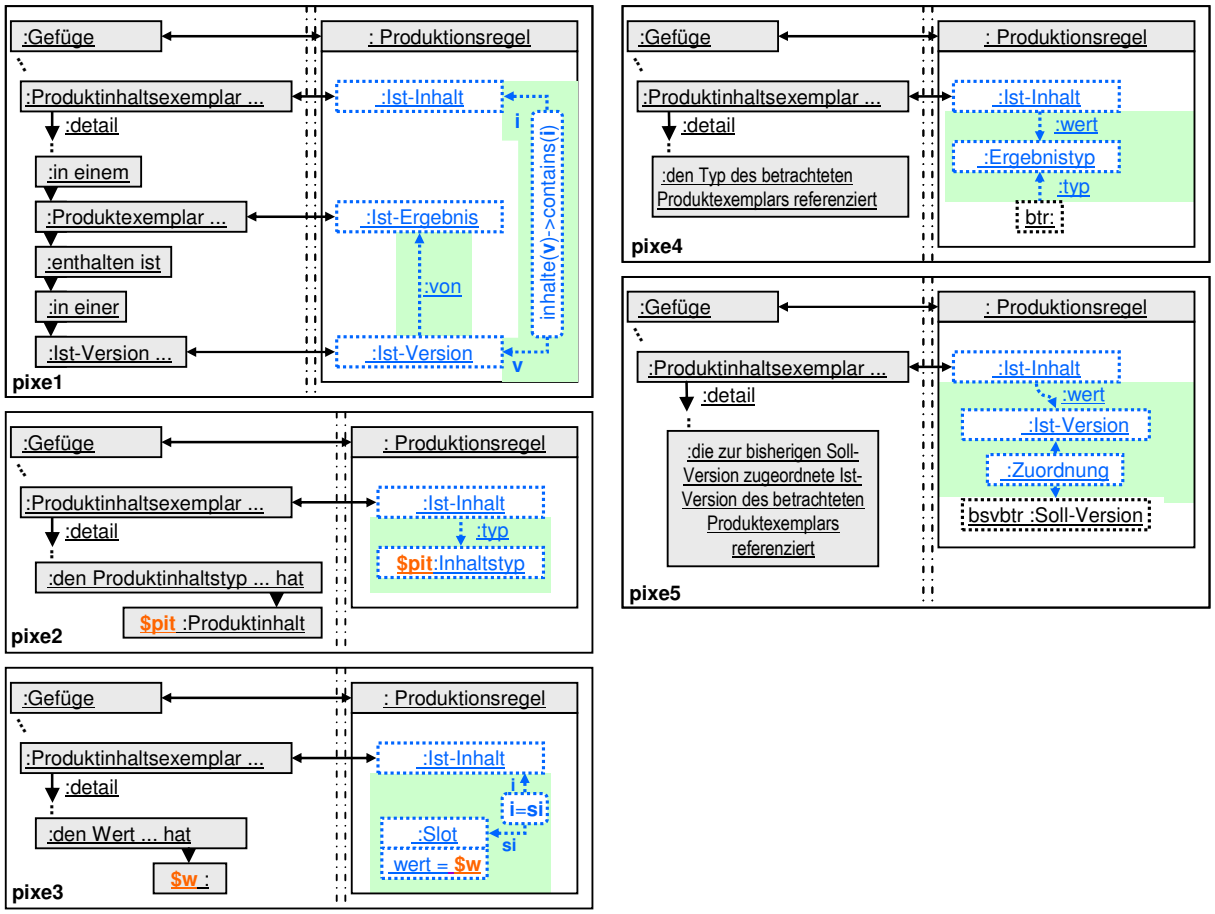


Abbildung 6.19 Produktionsregeln für Produktinhaltsexemplareinschränkungen

Abbildung 6.20 zeigt die Produktionsregeln für die Semantik von Meilensteineinschränkungen. Die Produktionsregel mse1 ist analog zu den anderen „fachlichen“ Typisierungen – wie etwa für Produktexemplare und Produktinhaltsexemplare. Bei mse2 wird der Betrachtete als eine Soll-Version angesehen, die zum übergeordneten Meilenstein eingeplant ist. Bei mse3 wird der dagegen der übergeordnet quantifizierte Meilenstein gleich dem Betrachteter gesetzt. Die Produktionsregel für mse4 kann nur im Zusammenhang mit Zustandsübergängen greifen, da nur diese in deren Eröffnng einen Suchmusterknoden mit der Id bsvbtr besitzen. Diese Id ist aus den Schablonen für die Versionsplanungen entnommen (siehe Abbildung 5.28) und repräsentiert die *b*isherige *S*oll-*V*ersion des für den auszuführen Zustandsübergang *betr*achteten Ist-Ergebnis. Bei mse5 wird der Typ des Betrachteten durch die zu diesem hinführende Kante vom Typ ist impliziert: Nach dem Produktbibliotheksmodell (und der Definition für deren mögliche Verfeinerungen) kommt nur eine Ist-Version als Belegung in Frage. In mse6 wird das Enthaltensein des Betrachteten als Inhalt in einer bestimmten Version gefordert – was als Technik auf ähnliche Weise bereits in pxe1 und pixe1 Anwendung fand.

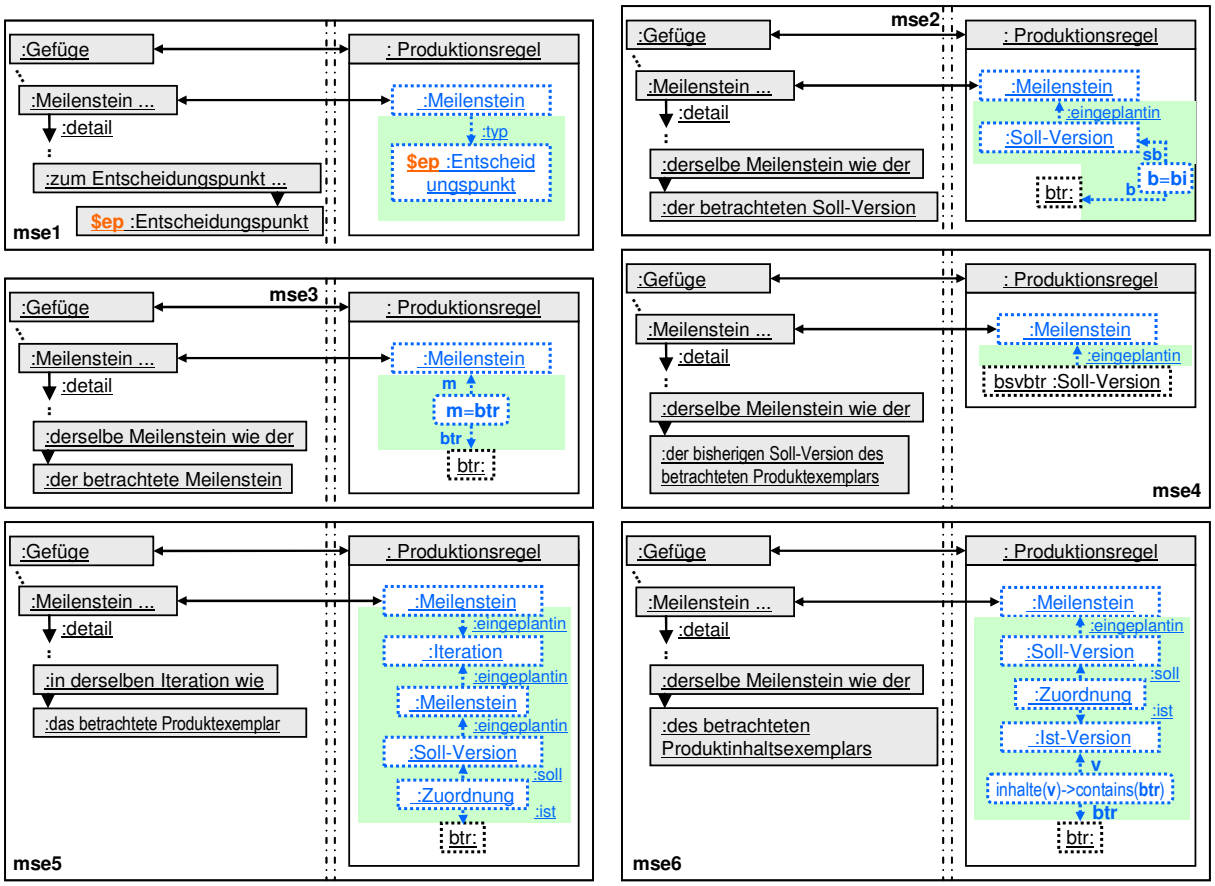


Abbildung 6.20 Produktionsregeln für Meilensteineinschränkungen

Abbildung 6.21 zeigt die Produktionsregel für die Semantik der (hier einzigen) Soll-Version-Einschränkung. Diese ist analog zu den anderen „fachlichen“ Typisierungen – wie etwa für Produktexemplare, Produktinhaltsexemplare und Meilensteine.

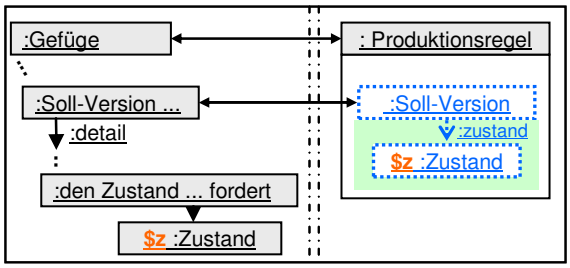


Abbildung 6.21 Produktionsregeln für Soll-Version-Einschränkungen

Abbildung 6.22 zeigt die Produktionsregeln für die Semantik von Ist-Version-Einschränkungen. Diese sind, anders als die bisherigen Einschränkungs-Produktionsregeln, nicht von einander unabhängig, sondern bilden selbst einen Zyklus aus Quantifizierung und Einschränkung. Der Grundstein wird zunächst mit ive1 oder ive2 gelegt. Die Ist-Versionen werden anhand deren Zuordnung zu Soll-Versionen und an diesen Soll-Versionen hängenden Einschränkungen identifiziert (vgl. Abbildung 5.6). Bei ie1 wird eine Soll-Version – die gesuchte beliebige – quantifiziert. Bei ie2 wird die gesuchte aktuellste quantifiziert (Suchmusterknoten v) sowie eine weitere Soll-Version (Suchmusterknoten vn) in einem Negationsbereich. Letztere wird sicherstellen, dass es für eine Belegung von v keine aktuellere Soll-Version mit den gleichen Eigenschaften gibt. Mit ive3 wird die Einschränkung des Zustands realisiert. Diese Regel greift sowohl für den von ive1 aufgestellten Suchmusterknoten v, als auch für die von ive2 aufgestellten Suchmusterknoten v und vn. Die Produktionsregeln ive4 bis ive6 sind ausschließlich in Kombination mit ive2 anwendbar. In

ive4 werden die beiden quantifizierten Soll-Versionen an denselben Meilenstein, in ive5 an dieselbe Iteration, und in ive6 an dieselbe Iteration mit bestimmten Eigenschaften gebunden. Dabei muss der Negationsberich, in dem vn liegt, nicht erweitert werden – um beispielsweise die Kante vom Typ eingeplantin mit aufzunehmen. Denn wegen der – zumindest in diesem Szenario vorhanden – Symmetrie können die entsprechenden Graphelemente mit den gleichen Graphelementen belegt werden, wie deren Gegenstücke bei v. Dies gilt auch für die durch ive3 ggf. erzeugte Zustandseinschränkung.

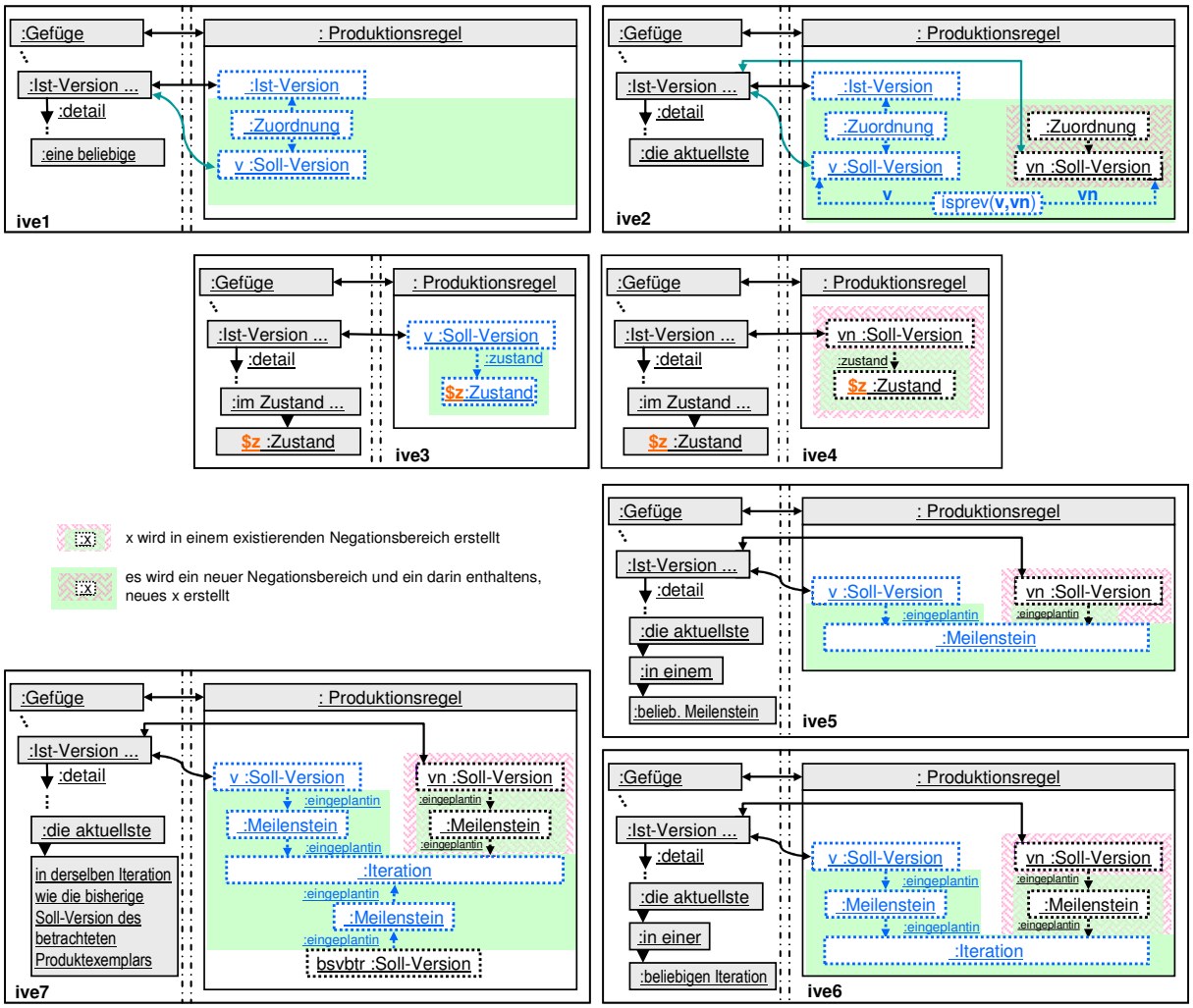


Abbildung 6.22 Produktionsregeln für Ist-Version-Einschränkungen

Abbildung 6.23 zeigt die Produktionsregeln für den Abschluss von Produktionsregeln. Mit ea2 werden bei erzeugenden Abhängigkeiten die Quantoren an den Betrachteten und den Meilenstein aus der Eröffnung gebunden. Durch zü2 bis zü4 wird die Negation in der Gefügekette umgesetzt. Dazu sind sämtliche Graphelemente bzw. Negationsbereiche die durch die vorherigen Produktionsregelarten erzeugt wurden, in einen gemeinsamen Negationsbereich aufzunehmen. Da dies nicht mehr mit einer einfachen Notation darstellbar ist, wird das Virtualisierungsmodell direkt genutzt: Durch zü2 wird zunächst der gemeinsame Negationsbereich mit der Id allin erzeugt. Zu diesem werden durch zü3 alle erzeugten Negationsbereiche hinzugefügt. Durch zü4 werden noch diejenigen Graphelemente hinzugefügt, die weder in einem Negationsbereich liegen noch btr oder bsvbtr als Id haben. Da in einem Negationsbereich liegende Graphelemente nicht an der Ignorierung teilnehmen, wird mit zü5 die entsprechende Verbindung entfernt.

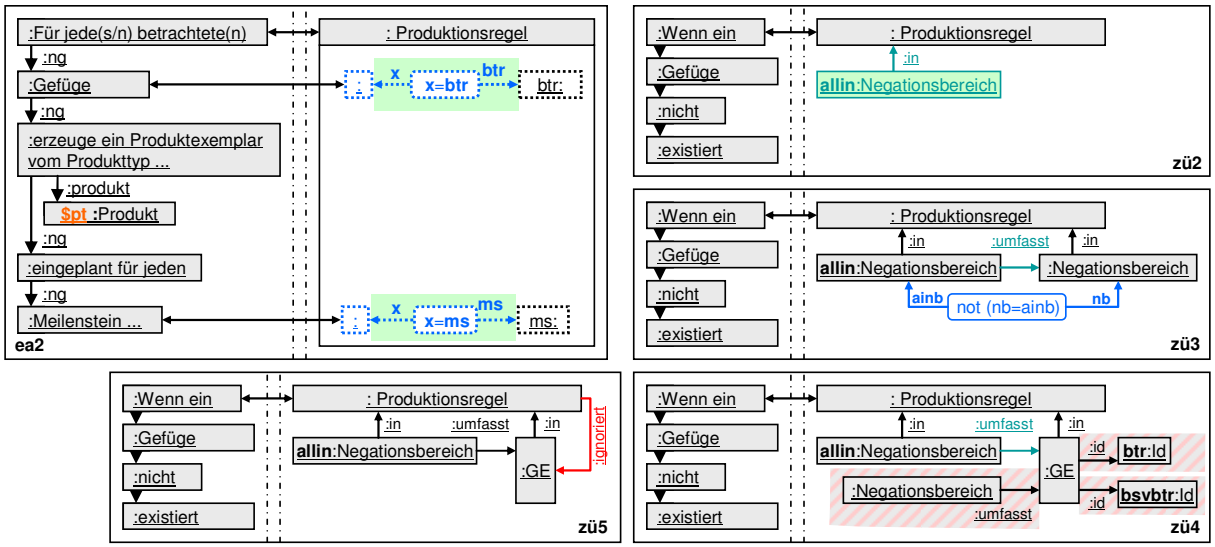


Abbildung 6.23 Abschliessende Produktionsregeln

Mit PBS⁺ beschriebenes VM_0: VM^+

Da PBS^+ im Wesentlichen eine Erweiterung von PBS_0 ist, können die meisten Elemente von VM_0 zu VM^+ übernommen werden. Konkret umfasst dies Produkte und Themen, Produkttypen, Vorgehensbausteine, Projektmerkmale und –werte, Entscheidungspunkte und Projektdurchführungsstrategien mit deren Bestandteilen, sowie schließlich auch die informalen erzeugenden Abhängigkeiten und Tailoringabhängigkeiten. Im Folgenden wird gezeigt, was durch die gegenüber PBS_0 neuen Konstrukte von PBS^+ in VM^+ gegenüber VM_0 neu hinzukommt. Konkret sind dies Produktinhalte, Zustandsmodelle und Gefügeketten zur automatisierung von Planungs- und Steuerungsvorgaben sowie Zustandsübergangs-bedingungen.

Abbildung 6.24 zeigt die hinzugefügten Produktinhalte in Beziehung zu den Themen und Produkten, die sie formalisieren. Für die Dekomposition der Systemarchitektur werden alle identifizierten Einheiten über den gleichnamigen Produktinhalt abgebildet. Die Festlegung, ob es sich dabei um SW- oder HW-Einheiten handelt, wird über entsprechend benannte Produktinhaltsreferenzen formalisiert, die über ihre Referenz als Prädikate wirken. Gleiches gilt für das Thema der zu spezifizierenden Systemelemente, in der die Produktinhaltsreferenz zuSpezifizierendeEinheit ebenfalls als Prädikat auf den identifizierten Einheiten angelegt ist. Für das QS-Handbuch wird eine Produktreferenz verwendet, die im Rahmen des verfeinerten Produktbibliotheksmodells für einen Ergebnistyp-Link steht. Als letzter besonderer Produktinhalt ist der für das Thema Prüfobjekt des Prüfprotokolls zu nennen, der im Rahmen des Produktbibliotheksmodells für einen Ist-Ergebnis-Link steht.

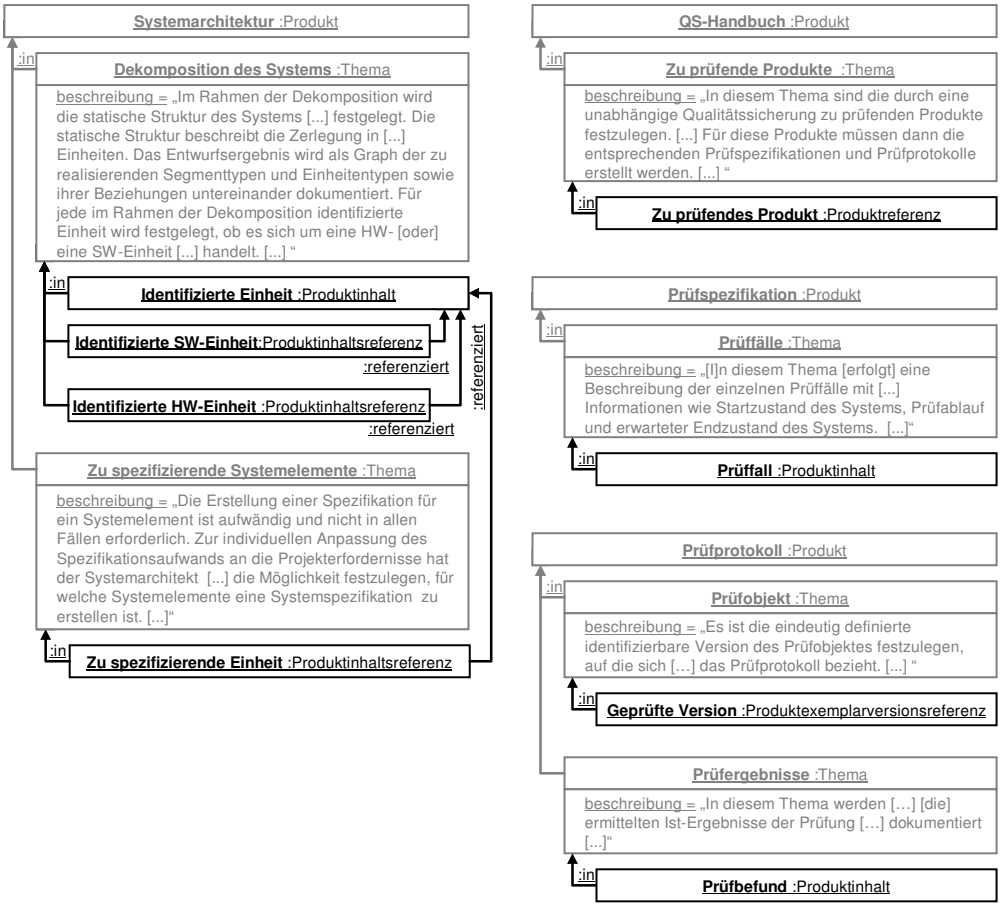


Abbildung 6.24 Produktinhalte von VM^+

Abbildung 6.25 zeigt die Zustandsmodelle von VM+. Dabei handelt es sich um ein allgemeines Zustandsmodell sowie ein spezielles für Prüfprodukte. Letzteres schließt eine endlose Rekursion der Prüfung aus. D.h. selbst wenn beispielsweise das Produkt Prüfprotokoll im QS-Handbuch als zu prüfen aufgenommen gelistet wäre, würde durch das spezielle Bearbeitungszustandsmodell dennoch immer nur eine Eigenprüfung notwendig sein.

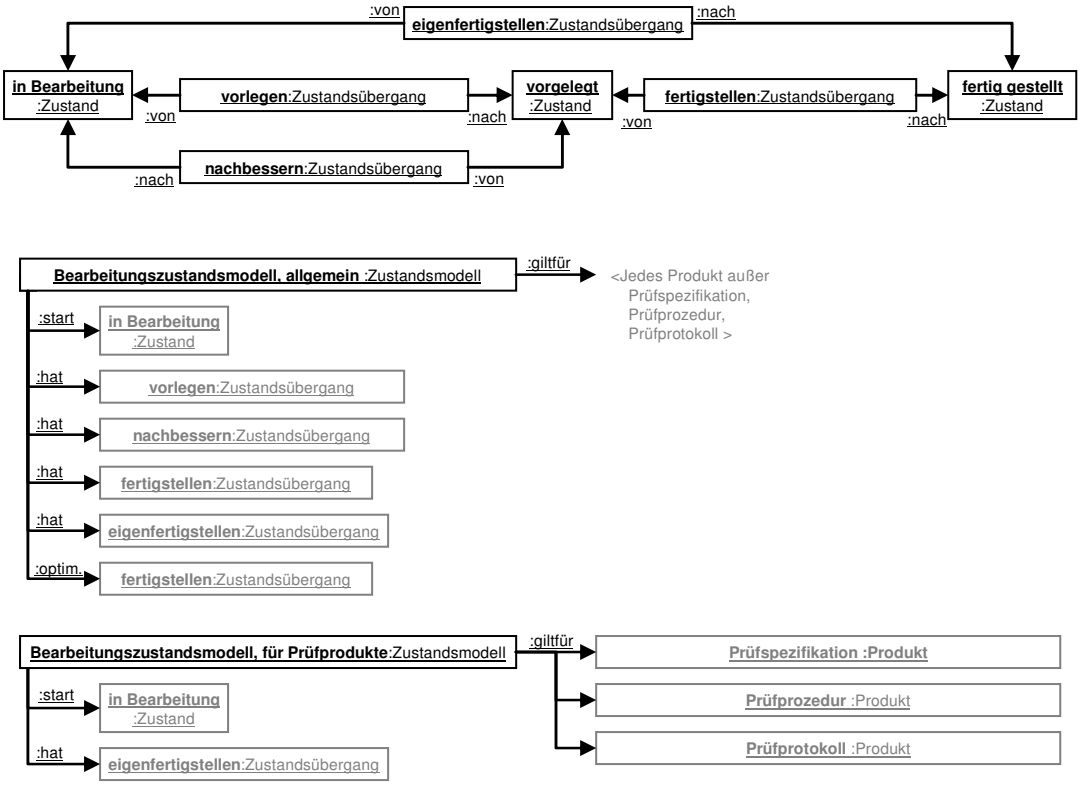


Abbildung 6.25 Zustandsmodelle von VM+

Abbildung 6.26 zeigt die mit Gefügeketten formalisierten Zustandsübergangsbedingungen, die sich aus der Instanz 1.4.5 Qualitätssicherung und Produktzustandsmodell ergeben. Dargestellt werden jedoch nur zwei der vier Zustandsübergangsbedingungen, da sich jeweils zwei stark ähnelt: Die Zustandsübergangsbedingung für den Zustandsübergang vorlegen unterscheidet sich von der des Zustandsübergangs eigenfertigstellen nur durch die Abwesenheit des vorvorletzten Terminals nicht. Die Zustandsübergangsbedingung für den Zustandsübergang fertig stellen unterscheidet sich von der des Zustandsübergangs nachbessern nur darin, dass der Wert erfolgreich anstelle von nicht erfolgreich referenziert wird. Die Gefügekette für den Zustandsübergang nachbessern zeigt die Verwendung von Phrasen, wobei hier die Identifikation eines bestimmten Zustands abgekürzt wird.

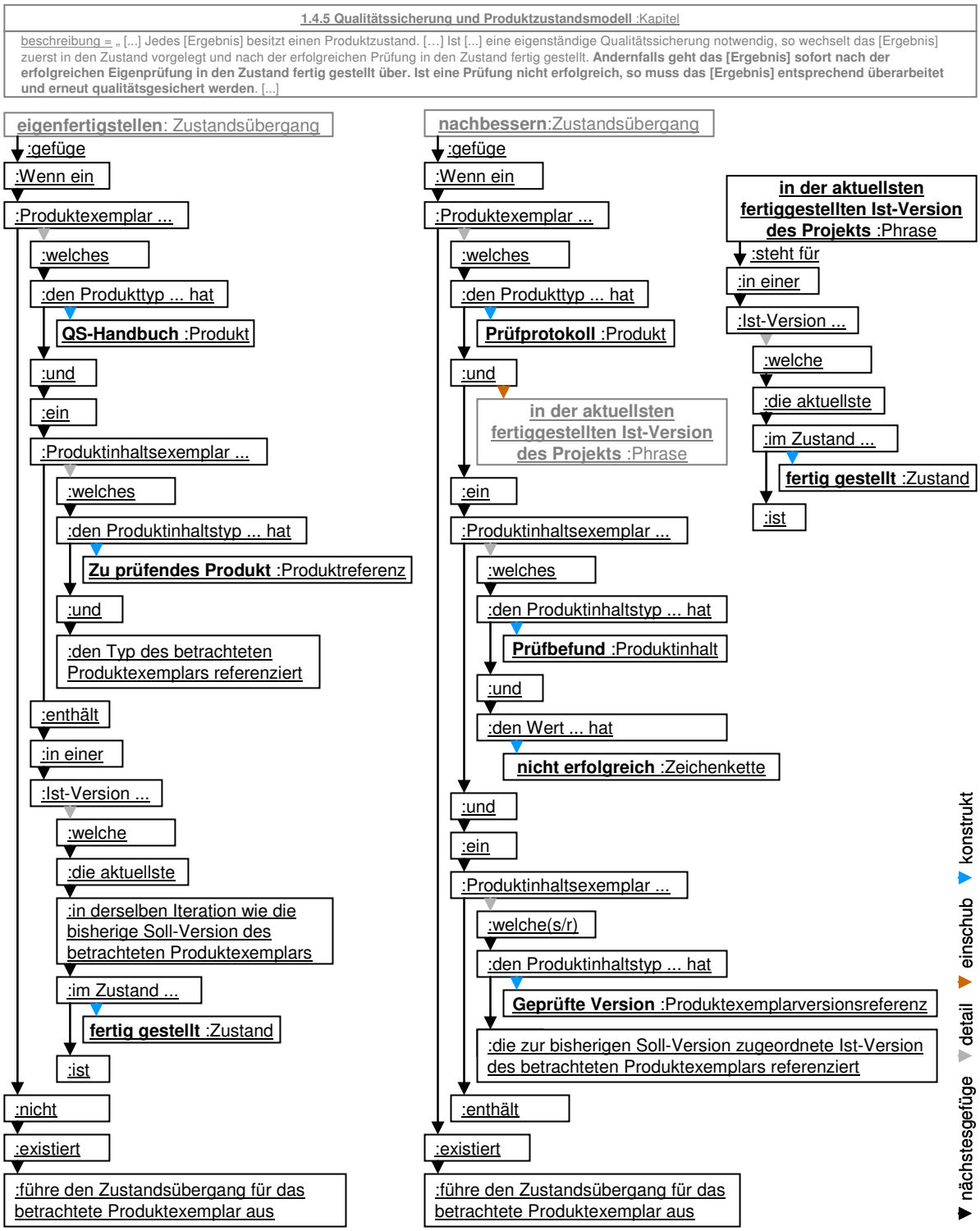


Abbildung 6.26 Beispiele von formalisierten Zustandsübergangsbedingungen in VM_0

Abbildung 6.27 zeigt Gefügeketten für erzeugende Abhängigkeiten und Tailoring-abhängigkeiten. Da beide auf der Existenz von identifizierten SW-Einheiten beruhen, kann hier eine entsprechende Phrase zur Wiederverwendung gewählt werden.

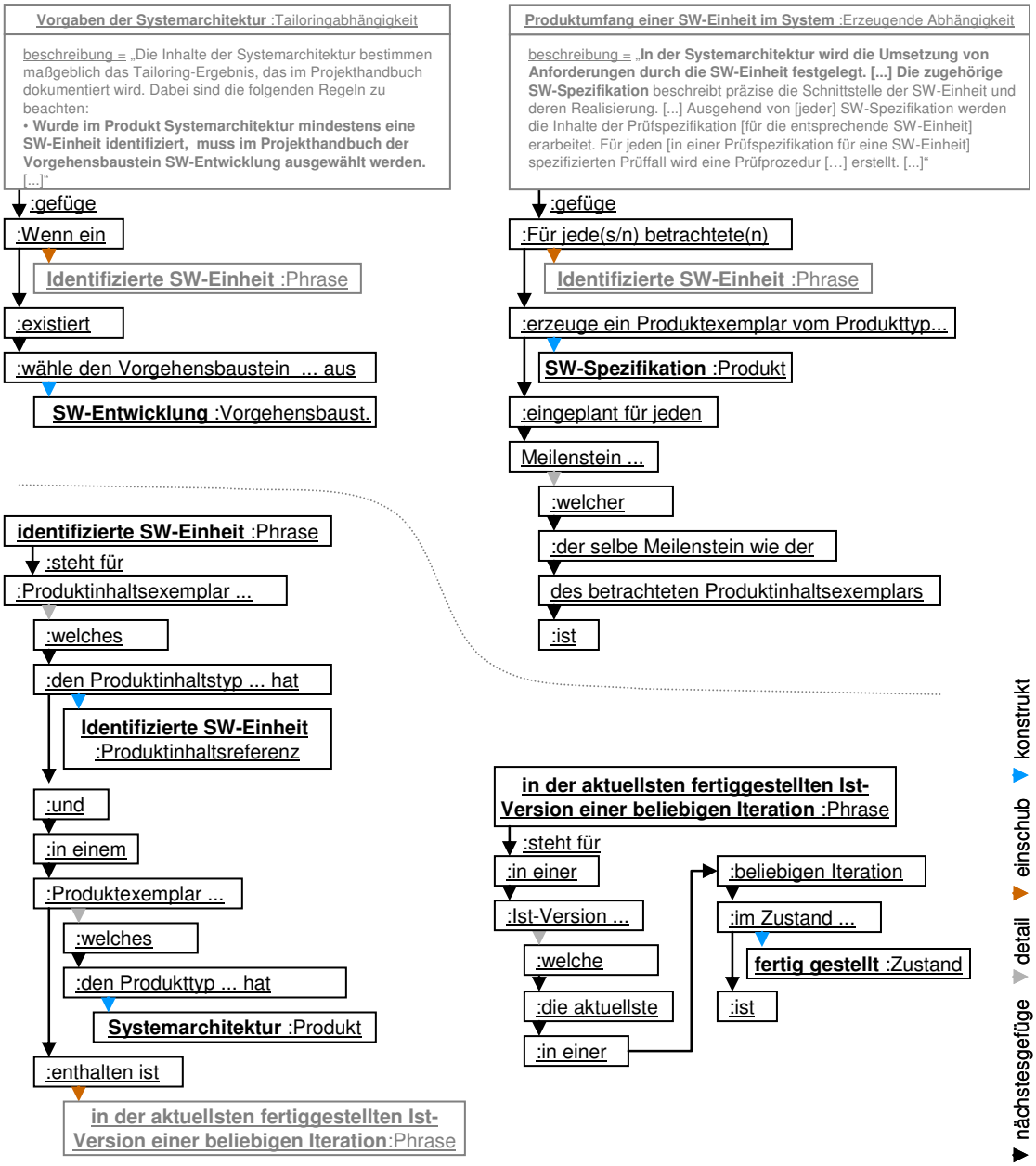


Abbildung 6.27 Beispiele von formalisierten Abhängigkeiten (1/3)

Abbildung 6.28 zeigt Gefügeketten für zwei weitere erzeugende Abhängigkeiten, wobei diesmal die Quantifizierung über Soll-Versionen und Meilensteinen Anwendung findet. Abbildung 6.29 zeigt wie Produktexemplare quantifiziert werden und dass an eine erzeugende Abhängigkeit auch mehrere Gefügeketten gehängt werden können.

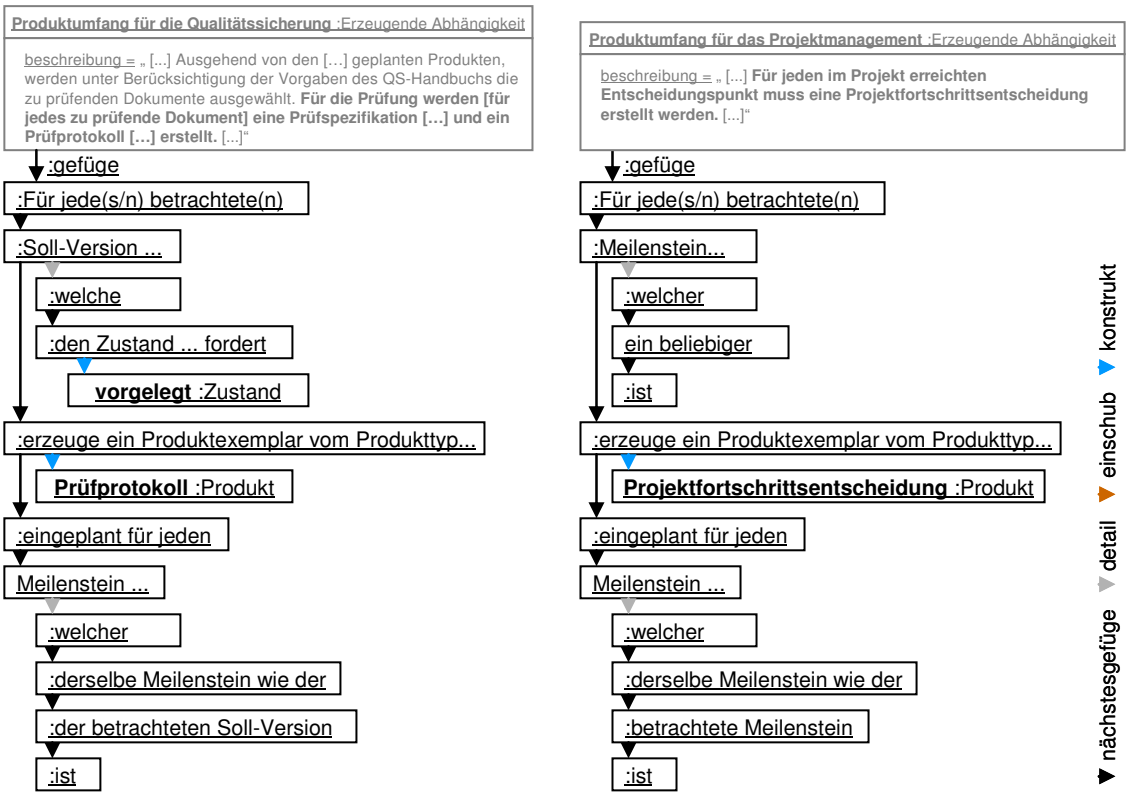


Abbildung 6.28 Beispiele von formalisierten Abhängigkeiten (2/3)

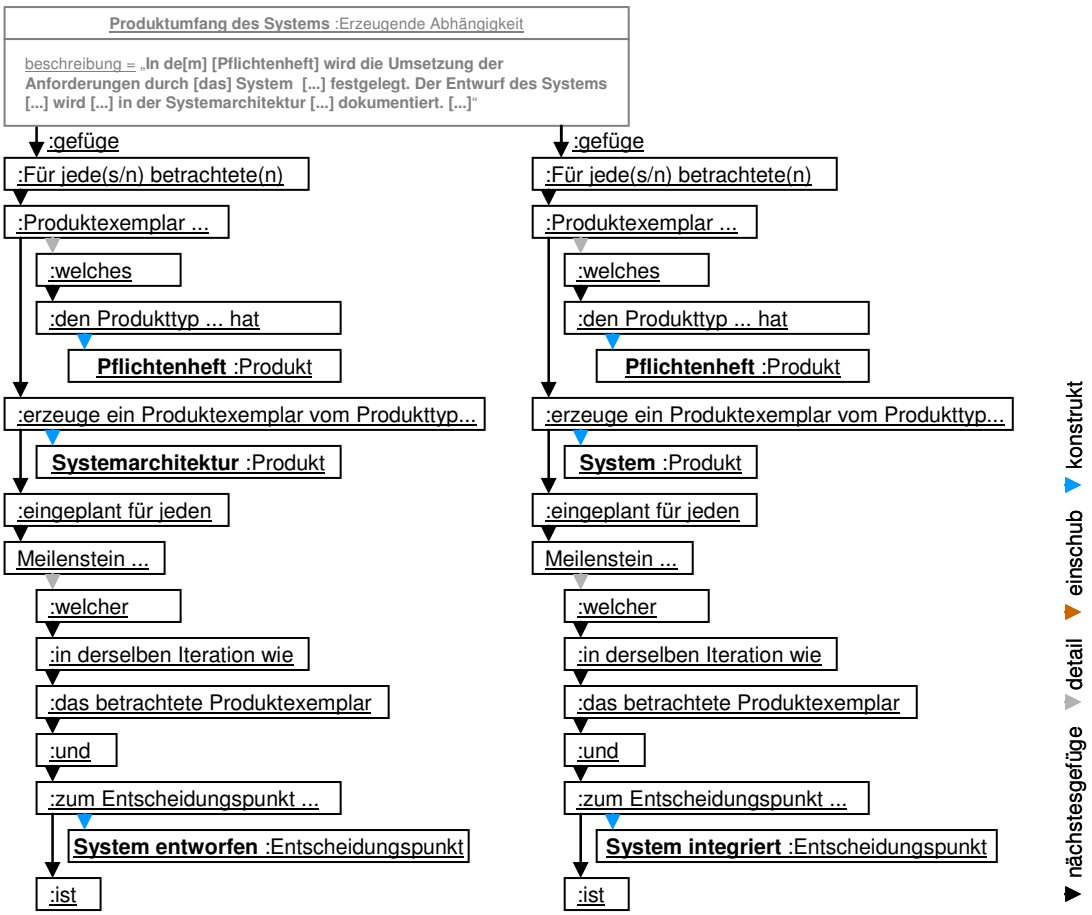


Abbildung 6.29 Beispiele von formalisierten Abhängigkeiten (3/3)

Die Formalisierung der übrigen erzeugenden Abhängigkeiten aus VM_0 verläuft analog zu den bisher gezeigten Beispielen. So ist der Produktumfang einer SW-Einheit im System um eine

weitere Gefügekette zur Einplanung der SW-Einheiten für den nächsten Meilenstein zum Entscheidungspunkt Systemelemente realisiert zu ergänzen. Auf gleiche Weise können auch die Prüfspezifikationen eingeplant werden, wobei dafür das Prädikat zu Spezifizierende Einheit anstelle von identifizierte SW-Einheit heranzuziehen ist. Wird als Produkt nicht die Systemarchitektur, sondern die Prüfspezifikation betrachtet, ergibt sich die Gefügekette zur Einplanung von Prüfprozeduren. Lieferung- und Abnahmedokumente sind dagegen analog zur Einplanung der Systemarchitektur aus Abbildung 6.29. Die Semantikabbildung stellt dabei sicher, dass nur für relevante Produkte Soll-Ergebnisse, und nur für nicht-externe Produkte Soll-Versionen eingeplant werden.

6.2 Semantik von VM+

Dieses Kapitel zeigt die Semantik von VM+, die durch Anwendung der Semantikabbildung entsteht. Da die meisten Produktionsregeln der Semantikabbildung unmittelbar eine Produktionsregel für die Produktbibliothek erzeugen, wird hier anhand von Beispielen nur der Teil zur Verarbeitung der Gefügeketten betrachtet, bei dem eine umfangreichere Berechnung erfolgt, bis eine Produktionsregel für die Semantik entsteht.

6.2.1 Ermittlung der Semantik

Abbildung 6.30 zeigt die Berechnung der Semantik der Tailoringabhängigkeit aus Abbildung 6.27. Dargestellt ist sowohl das nach Anwendung der Semantikabbildung entstehende Ergebnis, als auch – grau an der Seite – die verwendeten Produktionsreglen und deren Reihenfolge. Die Reihenfolge ist hier jedoch keine totale Ordnung. Daher werden vier Schritte angegeben, für die folgendes gilt: Die Reihenfolge aller Regelanwendung innerhalb eines Schrittes ist beliebig. Um den nächsten Schritt (n+1) zu beginnen, müssen alle Regelanwendungen im vorherigen (n) abgeschlossen sein.

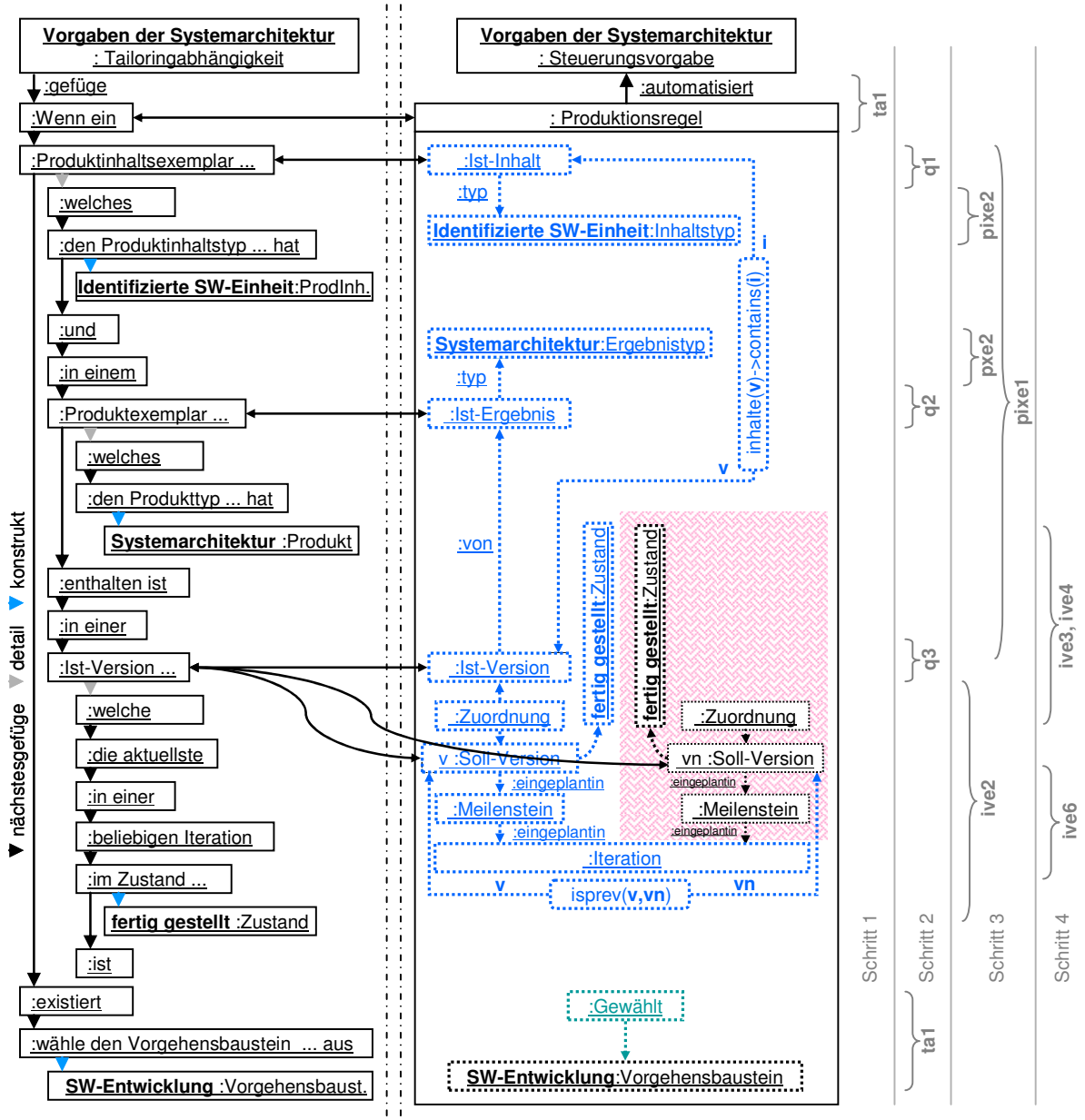


Abbildung 6.30 Berechnung der Semantik einer Tailoringabhängigkeit aus VM+

Abbildung 6.31 zeigt die mit der Semantikabbildung ermittelten Semantiken für die erzeugende Abhängigkeit aus Abbildung 6.27 (oben-rechts) und die beiden Zustandsübergangsbedingungen aus Abbildung 6.26.

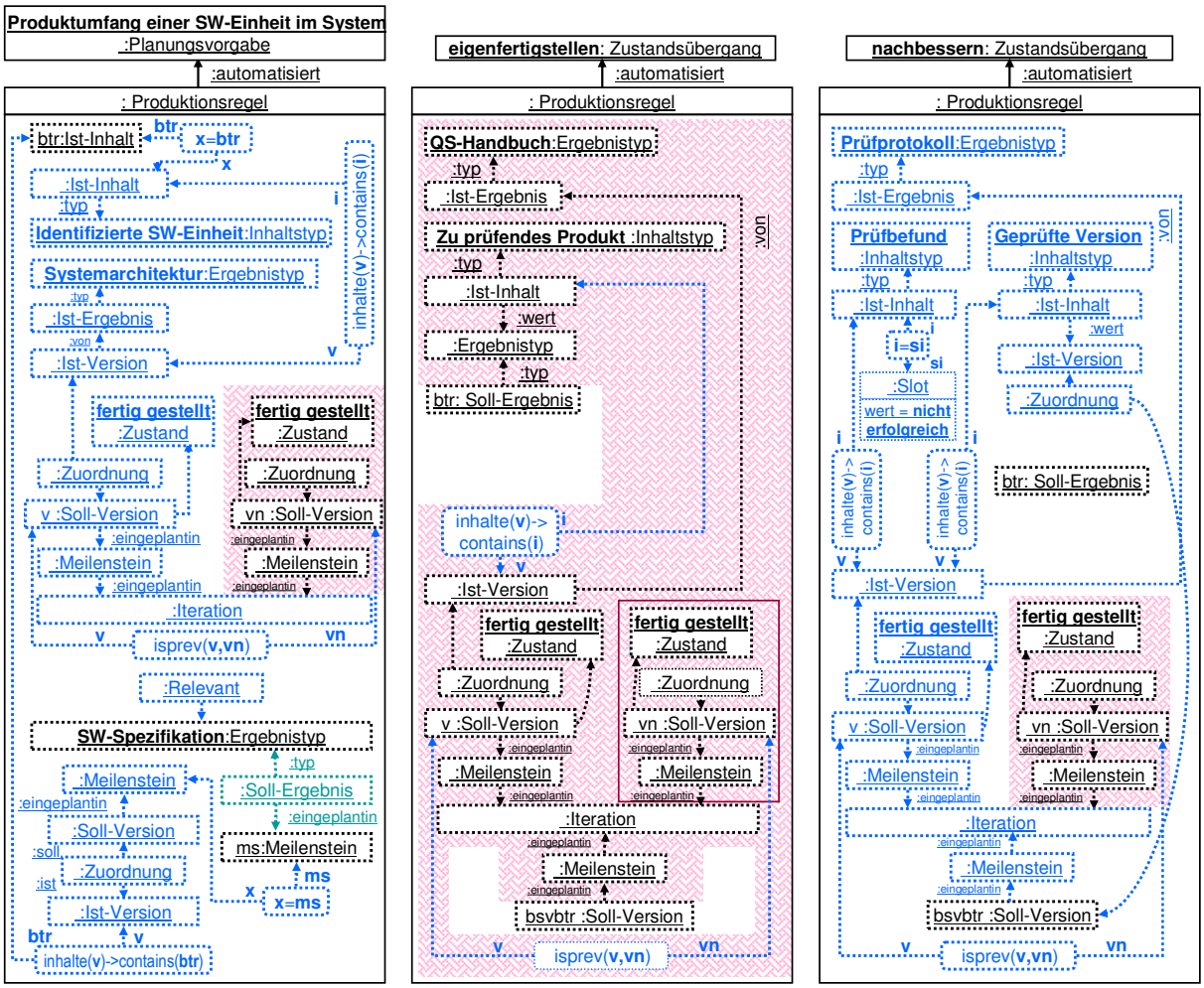


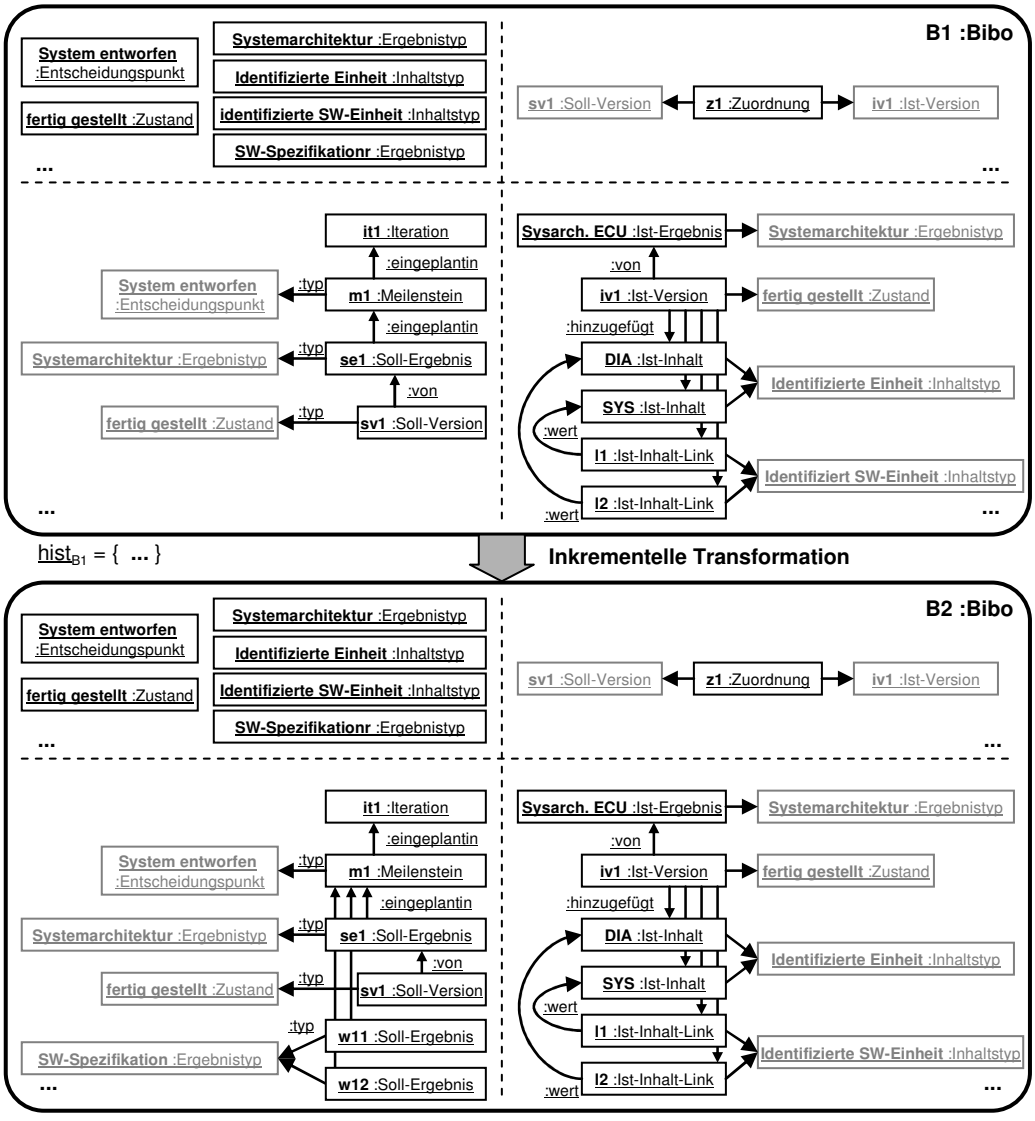
Abbildung 6.31 Beispielsemantik für erzeugende Abhängigkeit und Zustandsübergänge

6.2.2 Anwendung der Semantik

Die Anwendung der Semantik entspricht der Ausführung der Funktionen planung (Kapitel 5.3.2) und steuerung (Kapitel 5.3.4) aus der Lösung. Da in beiden Fällen virtualisierte Produktionsregeln ausgeführt werden, wird dies nur an einem Beispiel vorgeführt. Verwendet wird dabei die Produktionsregel zur Erzeugenden Abhängigkeit des Produktumfangs einer SW-Einheit im System (vgl. Abbildung 6.31, links).

Für ein praxisbezogenes Szenario sei ein Projekt zur Entwicklung einer Electronic Control Unit (kurz ECU) für einer Freisprechanlage herangezogen (siehe [96]). Zu einem Zeitpunkt B1 (Abbildung 6.32) liegt in der entsprechenden Produktbibliothek u.a. eine eingeplante Systemarchitektur se1 die durch die Systemarchitektur Sysarch. ECU über die Zuordnung z1 und entsprechende Versionen abgedeckt ist. Identifiziert wurden die Einheiten DIA (Diagnose-Management) und SYS (System-Management), die durch I1 und I2 als SW-Einheiten qualifiziert wurden. Durch Anwendung der Produktionsregel aus Abbildung 6.31 entsteht die Produktbibliothek B2, in der durch w11 und w12 zwei (weitere) SW-Spezifikationen eingeplant sind. Zusätzlich wurde die bisherige Transformationshistorie um zwei entsprechende Einträge erweitert: p1 steht dabei für die devirtualisierte Produktionsregel die

hier angewendet wurde. Die vier Einträge ordnen den vier für die Transformationshistorie nicht zu ignorierenden Musterelementknoten die Belegungen zu. Von den entsprechenden Ids sind die ersten zwei, btr und ms bereits in der Abbildung 6.32 zu sehen – da sie dort über das Id-Patternmatching gezielt erzeugt, und entsprechend von Interesse waren. Die übrigen Ids sind bei der Anwendung der Semantikabbildung durch die Erzeugungsfunktion zufällig entstanden. Für dieses Beispiel werden zwei beliebig gewählte Ids verwendet. Weiterhin ist zu beachten, dass in Abbildung 6.32 der Ablauf effektiv darstellt ist – ohne die Effekte der Änderungsvirtualisierung explizit anzugeben.



$hist_{B2}$ = { (p1,{btr↦I1, ms↦m1, n93↦SW-Spezifikation, n93↦w11 }),
(p1,{btr↦I2, ms↦m1, n93↦SW-Spezifikation, n93↦w12 }), ... }

Abbildung 6.32 Beispiel der Anwendung der Semantik von VM+

6.3 Integrationsansatz für das Werkzeugumfeld des V-Modell XT

Die vier Schritte des Projektcontrollings werden in der Praxis durch Werkzeuge unterstützt ausgeführt. Beispielsweise sind MS Project [97] und SAP R/3 [98] zwei der bekannteren kommerziellen Vertreter für Planung und Kontrolle. Die Bedienkonzepte dieser und ähnlicher Werkzeuge sind den Anwendern gut vertraut. Für die Evaluation ergibt sich daher die Frage, ob sich die in Kapitel 5 vorgestellte Lösung in diese Bedienkonzepte harmonisch integrieren lässt – oder von den Anwendern ganz neue Fähigkeiten erfordert.

Die Auswahl der betrachteten Werkzeuge richtet sich hier nach der Werkzeugumgebung des V-Modell XT – zur Fortführung des Fallbeispiels VM_0. Entsprechend wird die Frage der Integrierbarkeit der Lösung für Planung und Kontrolle anhand von Gantt Project [99] in Verbindung mit dem V-Modell XT Projektassistent [100] behandelt (Kapitel 6.4.1), die Durchführung anhand von Eclipse [101] (Kapitel 6.4.2) und die Steuerung anhand des V-Modell XT Editors [102] (Kapitel 6.4.3). Die aufgezeigten Wege für eine harmonische Integration sind jedoch auch auf andere Werkzeuge übertragbar.

6.3.1 Planung und Kontrolle

Bisheriges Vorgehen. Die tägliche Planung und Kontrolle wird mit einem Werkzeug wie GanttProject begleitet. In Anlehnung an die Netzplantechnik (vgl. [103]) wird das Projekt in Meilensteine und Vorgänge (ggf. hierarchisch) gegliedert, die wiederum mit Zielterminen versehen, bezüglich des benötigten Zeitaufwands eingeschätzt, und mit Ressourcen abgedeckt werden. Sofern die einzuplanenden Vorgänge nicht ad-hoc, sondern nach einem Vorgehensmodell abgeleitet werden sollen, gibt es für das Beispiel des V-Modells mit dem V-Modell XT Projektassistenten eine Werkzeugunterstützung[47]. Wie Abbildung 6.33 zeigt, werden die Vorgaben zum statischen Tailoring (siehe Kapitel 3.3.2) sowie zur Erstellung von Projektdurchführungsplänen formalisiert. Ein Projektleiter kann damit einen initialen Projektdurchführungsplan erzeugen lassen, der im weiteren Verlauf des Projektes allerdings per Hand aktualisiert werden muss[48]. Dies betrifft sowohl Änderungen hinsichtlich des statischen Tailorings (siehe Kapitel 3.3.3) sowie generell den induktiven Ergebnisumfang (siehe Kapitel 3.3.4). Auch muss der Projektleiter sämtliche Soll-Versionen manuell einplanen und aktualisieren, was in der Praxis meist zu feingliedrig ist.

[47] Für das Vorgehensmodell Hermes [79] ist der Hermes PowerUser [104] ein zum Projektassistent vergleichbares Werkzeug.

[48] Da die entsprechenden Vorgaben nicht formalisiert sind, kann der Projektassistent nur für initiale Produkte Soll-Ergebnisse einplanen: Der Projektassistent erzeugt zwar, zusätzlich zu den initialen Produkten, pro Meilenstein auch genau eine Einplanung für jeden zum entsprechenden Entscheidungspunkt vorzulegenden Produkt. Allerdings müssen diese Einplanungen vom Projektleiter später dennoch manuell auf die tatsächliche Anzahl angepasst werden. Diese Einschränkung gilt dabei nicht nur für den Projektassistenten, sondern auch beispielsweise für den Hermes PowerUser.

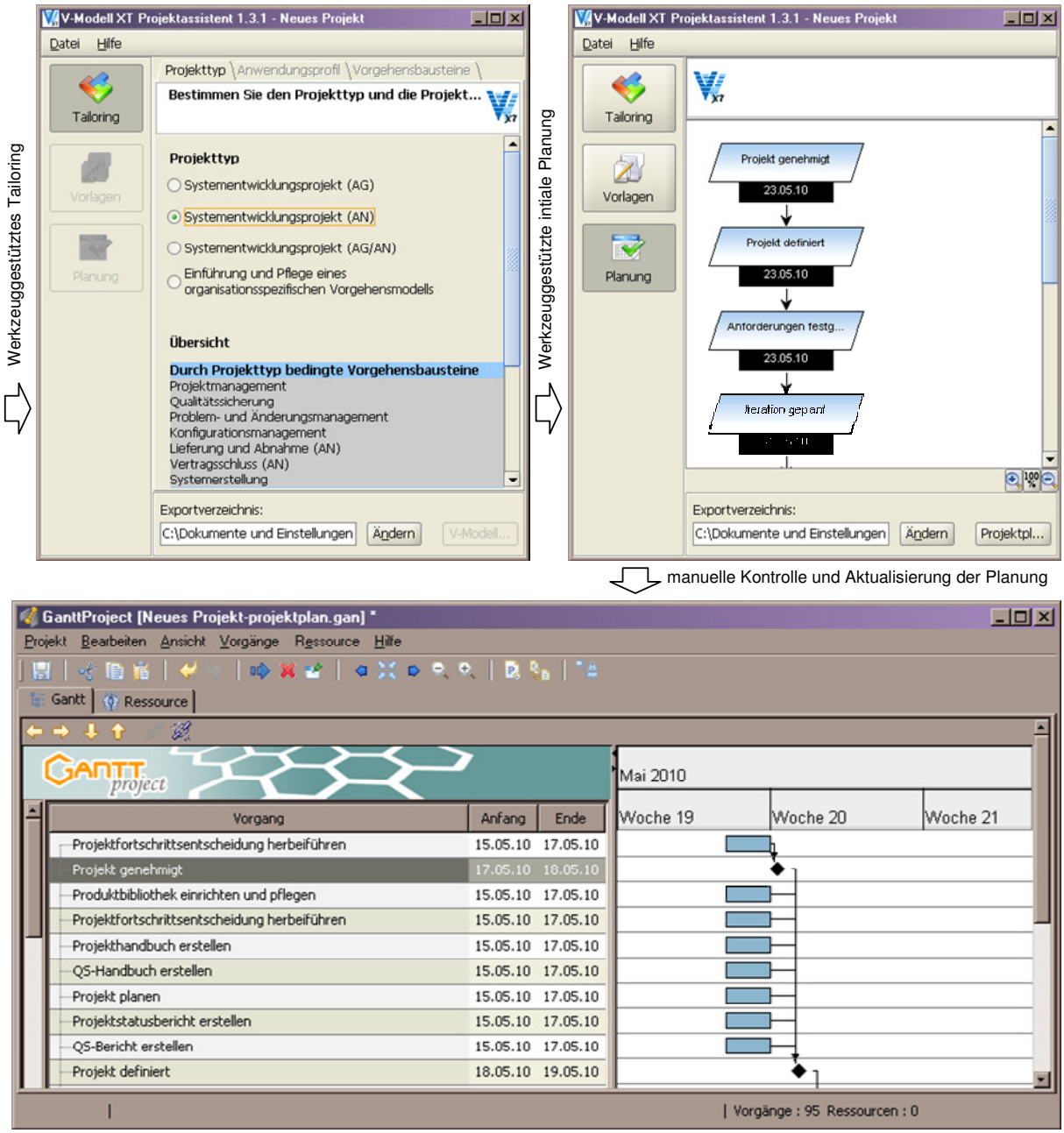


Abbildung 6.33 Werkzeugeinsatz zur Projektplanung und -kontrolle in der Praxis

Integration der Lösung in bisheriges Vorgehen. Für den Projektleiter hat die Anwendung der Lösung zunächst den positiven Effekt, die Aktualisierung der Soll-Ergebnisse und -Versionen allein durch einen Knopfdruck auszuführen. Die Zuweisung von Ressourcen, Zielterminen usw. bleibt davon unberührt. Durch die auf Ids basierte Aktualisierung können auch erzeugte Soll-Ergebnisse hierarchisch um- und geordnet werden. Um diesbezüglichen Aufwand bereits bei der Automatisierung zu minimieren, empfiehlt es sich, die erzeugten Soll-Ergebnisse nicht nach Meilensteinen zu ordnen (wie in Abbildung 6.33), sondern gemäß der erzeugenden Abhängigkeiten hierarchisch einzuplanen. Abbildung 6.34 zeigt das Vorgehen in Fortführung des Beispiels aus Abbildung 6.32. Jedes Soll-Ergebnis wird durch eine Produktionsregel erzeugt, die genau ein als Verursacher betrachtetes Element der Produktbibliothek (hier Knoten n91) hat. Um die übergeordnete Hierarchiestufe für die durch diese Produktionsregel neu erzeugten Soll-Ergebnisse w11 (bzw. w12) zu bestimmen, wird der violett markierte Weg verfolgt: Im ersten Schritt wird anhand der Transformationshistorie ermittelt, welches Bibliothekselement der als Verursacher zu betrachtende ist. In diesem Fall ist es der Ist-Inhalt l1 (bzw. l2) da dieser die Belegung des Suchmusterknotens mit der Id btr

ist. Im zweiten Schritt wird die zugehörige Ist-Version, und im dritten Schritt – über die Zuordnungen – die zugehörige Soll-Version ermittelt. Im vierten und letzten Schritt wird das Soll-Ergebnis bestimmt (hier se1) welches als die für w11 (bzw. w12) übergeordnete Hierarchie im Projektplan (Abbildung 6.34 rechts-unten) verwendet wird. Im Projektplan werden die Soll-Ergebnisse als Kombination von dessen Typ und Id dargestellt (Wahlweise kann der Typ auch stets als Teil der Id mitgeführt werden).

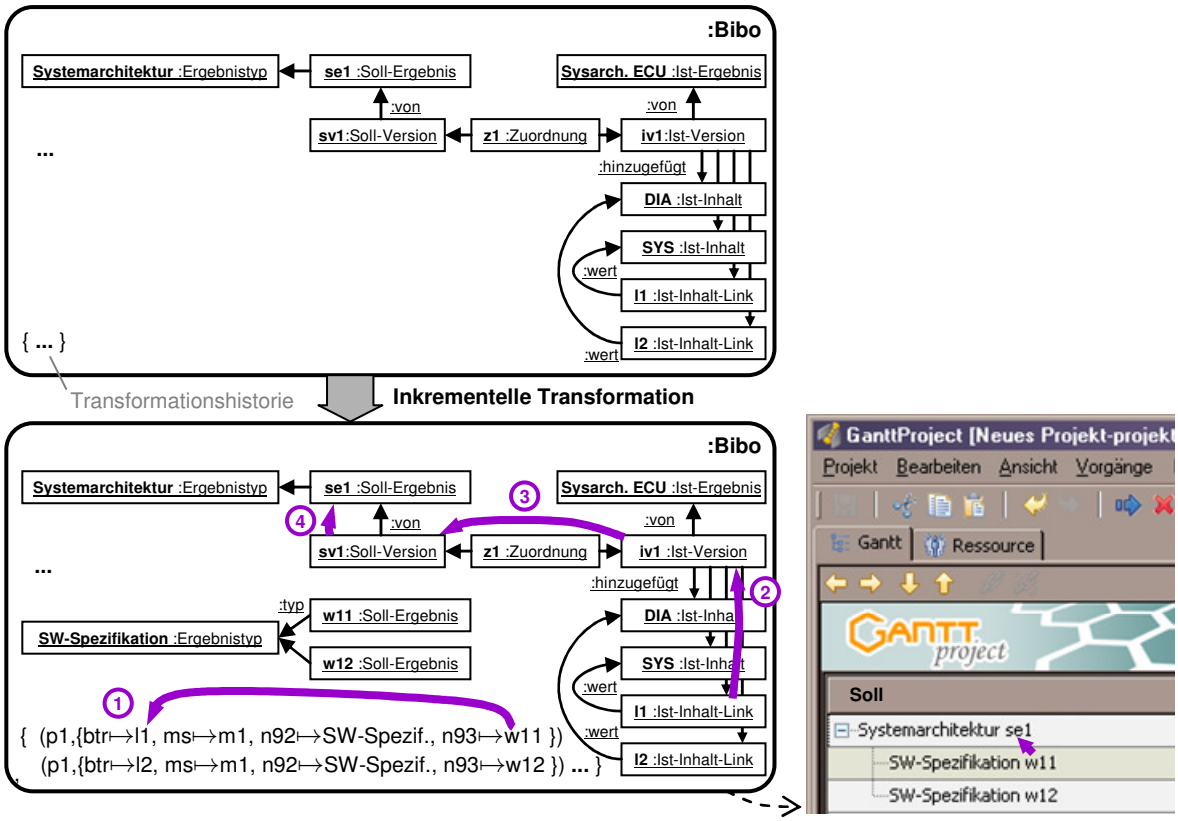


Abbildung 6.34 Planungshierarchien durch Auswertung Erzeugender Abhängigkeiten

Sofern ein Verursacher ein Produktexemplar ist, ist analog zu verfahren – wobei der Schritt (2) von einem Ist-Ergebnis (anstelle von einem Ist-Inhalt) ausgeht. Im Allgemeinen kann ein beliebiger Suchmusterknoten, dessen Belegung für die Transformationshistorie nicht zu ignorieren ist, als Verursacher und damit als die übergeordnete Hierarchie verwendet werden. Die Wahl dessen mit der Id btr ist also nur als ein Beispiel zu verstehen.

Um im Projektplan den Zusammenhang zwischen den Verursachern und den neu eingeplanten Elementen auch aus fachlicher Sicht herzustellen, können diese in Anlehnung an den Verursacher (bei Prädikaten das qualifizierte Element) benannt werden. Abbildung 6.35 zeigt, wie dies in Fortführung des Beispiels aus Abbildung 6.34 aussieht. Die Ids setzt sich dabei aus dem Typ, der Zeichenkette „für “ und der Id des Verursachers (hier des vom jeweiligen Verursacher I1 bzw. I2 qualifizierte Element SYS bzw. DIA) zusammen.

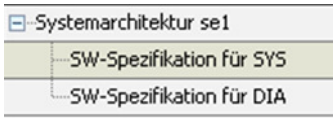


Abbildung 6.35 Nutzung der Verursacher-Ids zur Projektplan-Generierung

Für Meilensteine, Soll- und Ist-Versionen eignet sich dagegen eine andere Darstellung, da hier die Kontrolle hineinspielt. So müssen Ist- zu Soll-Versionen zugeordnet und diesbezüglich automatisch berechnete Zuordnungsvorschläge angeboten werden können. Dabei ist zu beachten, dass Ergebnisse auch über Meilensteingrenzen hinweg fortgeschrieben werden können. Aus diesen Eckpunkten resultiert die in Abbildung 6.36 gezeigte Organisation eines Projektplans:

- Versionen und Soll-Projektstände werden orthogonal zu den Soll-Ergebnissen angeordnet. Pro Soll-Ergebnis gibt es eine Zeile für Soll-Versionen (jeweils die obere) und eine Zeile für Ist-Versionen (jeweils die untere). Beispielsweise liegen für die Systemarchitektur im Meilenstein 6 drei Soll-Versionen und sechs Ist-Versionen vor, und drei weitere Soll-Versionen im Meilenstein 11 (vgl. Konstellation in Abbildung 5.10). Der Übersicht halber wird pro Version nur der Zustand, und in abgekürzter Form, dargestellt. Details können jedoch per Tooltip (vgl. [105]) eingesehen werden. Bei Soll-Versionen ist dies nur der geforderte Zustand. Bei Ist-Versionen kommen die Revisionsnummer und der Eincheckkommentar (bei Verwendung eines Repositories) sowie Datum und Urheber hinzu.
- Schwarz gefärbte Ist- und Soll-Versionen stellen zugeordnete Versionen dar, wobei eine schwarz gefärbte Kante die geltende Zuordnung angibt. Grün gefärbte Ist-Versionen und orange gefärbte Soll-Versionen sind solche, die noch nicht zugeordnet sind. Grau gefärbte Ist-Versionen sind solche, die bei der Berechnung von Zuordnungsvorschlägen ignoriert werden. Orange gefärbte Linien (wie die zwischen der ersten Soll- und Ist-Version der SW-Spezifikation für SYS) stellen Zuordnungsvorschläge dar.
- Zueinander zugeordnete Ist- und Soll-Versionen werden bündig untereinander dargestellt. Alle nicht zugeordneten Ist-Versionen werden in dem Meilenstein dargestellt, in dem die nächste, nicht zugeordnete Soll-Version ist.

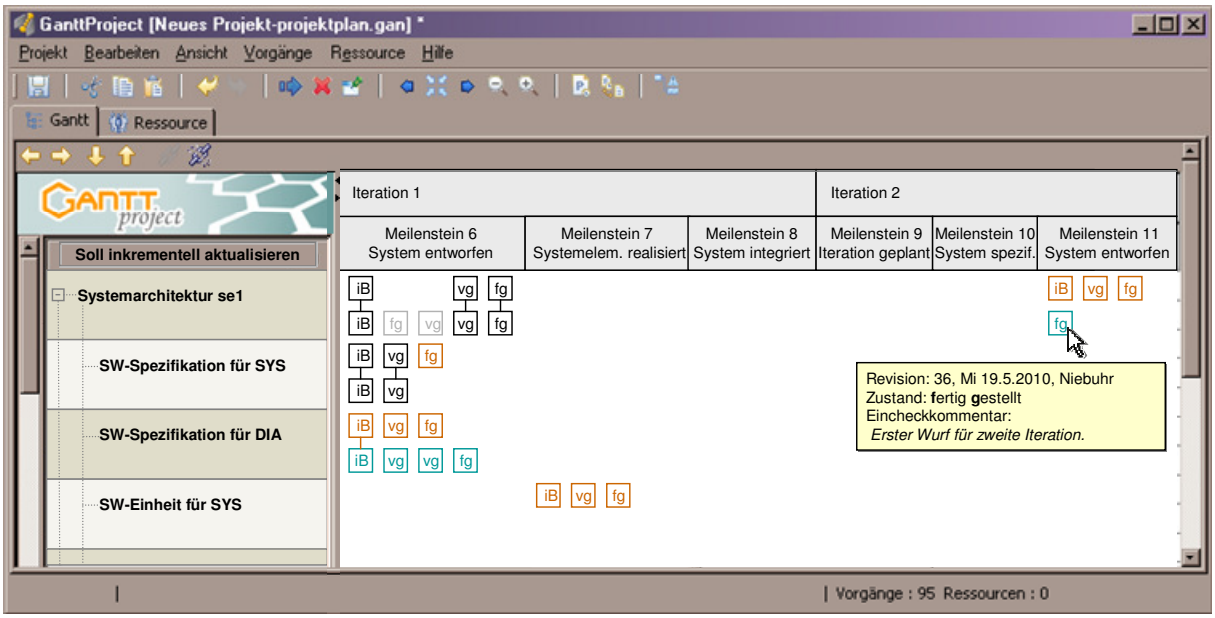


Abbildung 6.36 Organisation des Projektplans für das Automatisierungskonzept

Wie am Beispiel der SW-Spezifikation für SYS zu sehen, werden Zuordnungsvorschläge jeweils nur für die nächste noch nicht zugeordnete Soll-Version angezeigt. Dadurch können auch Fehler schnell entdeckt werden: Für die noch nicht zugeordnete Ist-Version der Systemarchitektur mit Revision 36 wird kein Zuordnungsvorschlag angeboten, obwohl Soll-Versionen noch offen stehen.

Korrekturmaßnahmen können, wie Abbildung 6.37 zeigt, als Kontextmenü angeboten werden. Neben dem Ignorieren der Version kann der Zustand auch korrigiert werden – z.B. durch das Ignorieren der aktuellen und das Erzeugen einer neuen Version, mit dem von der nächsten nicht zugeordneten Soll-Version erwarteten Zustand. Um Komplikationen mit dem eigentlichen Bearbeiter des Ergebnisses zu umgehen, kann dieser auch einfach nur über den vorliegenden Fehler informiert werden. Technisch könnte dies auch als eine automatische Korrektur beim nächsten Eincheckvorgang realisiert werden.

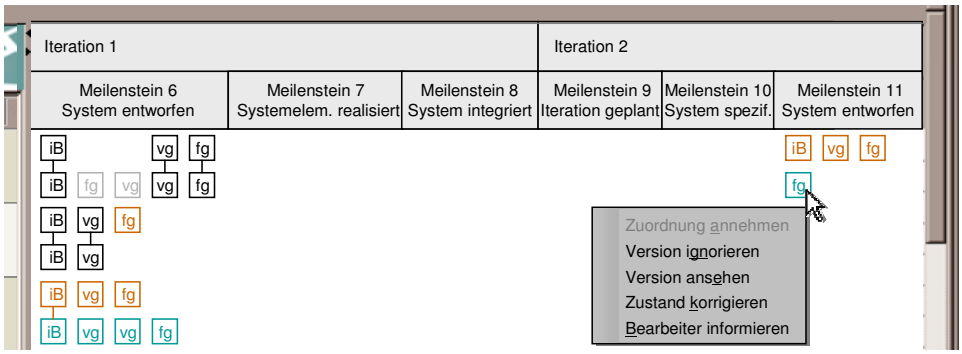


Abbildung 6.37 Korrekturmaßnahmen für die Kontrolle

Um der Konstellation vorzubeugen, dass eine erste Ist-Version zu mehreren Soll-Ergebnissen passt (vgl. Abbildung 5.8) kann in den Projektplan die Erzeugung der ersten Ist-Version auch proaktiv, aus dem Projektplan heraus, vollzogen werden. Dies kann entweder individuell (wie Abbildung 6.38 zeigt) für einzelne Soll-Version erfolgen, oder auch global für alle ersten Soll-Versionen aller Soll-Ergebnisse.

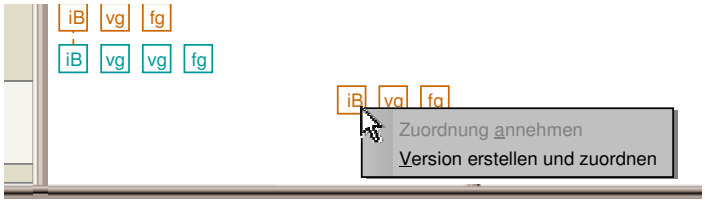


Abbildung 6.38 Automatische Erzeugung einer initialen Version

6.3.2 Durchführung

Bisheriges Vorgehen. Für die Erarbeitung der Ergebnisse werden in der Praxis eine Fülle unterschiedlichster Werkzeuge und Datenformate verwendet. So kommen zur Beschreibung von Anforderungen beispielsweise MS Word oder Doors [106] zum Einsatz. Spezifikationen und Architekturen werden mit Modellierungsumgebungen wie beispielweise Innovator [107] oder Eclipse erstellt. Für Systemelemente und Prüfprozeduren finden Entwicklungsumgebungen wie MS Visual Studio [108], ebenfalls Eclipse oder auch einfach nur Texteditoren wie ConText [109] ihre Anwendung. All diese Werkzeuge hinsichtlich ihrer Bedienung einzeln zu betrachten würde den Rahmen sprengen. Allerdings lässt sich die Auswirkungen der in Kapitel 5 vorgestellten Lösung auch anhand eines Beispiels allgemein skizzieren. Dazu wird mit Abbildung 6.39 die Modellierung des Themas Dekomposition einer Systemarchitektur durch ein UML-Komponentendiagramm betrachtet.

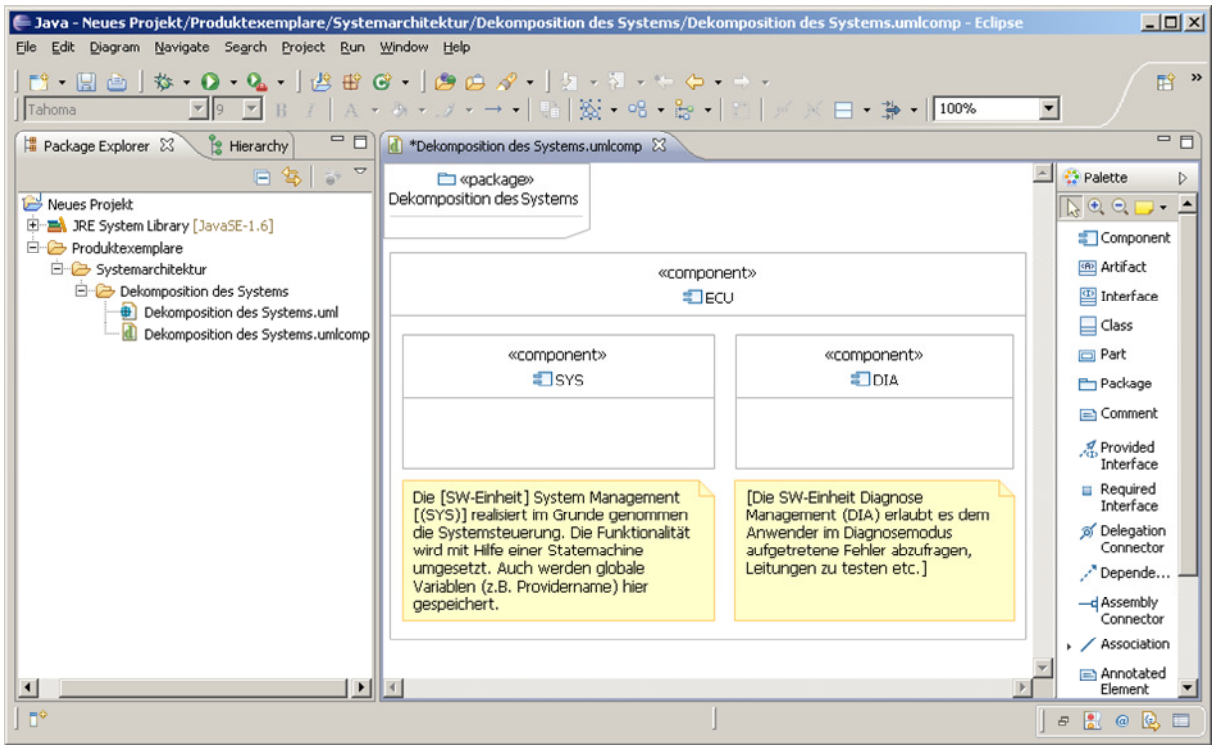


Abbildung 6.39 Werkzeugeinsatz zur Projektdurchführung in der Praxis

Integration der Lösung in bisheriges Vorgehen. Um die Lösung für die Operationalisierung des Projektcontrollings in der Praxis anzuwenden, müssen die Inhalte erarbeiteter Ist-Ergebnisse automatisiert ausgewertet werden können. Die in der Praxis verwendeten Formate sind meist jedoch nicht darauf ausgelegt, die von einem Vorgehensmodell beschriebenen Inhalte als solche auszuzeichnen. Beispielsweise ist anhand des UML-Komponentendiagramms der Abbildung 6.39 formal nicht erkennbar, ob es sich bei den Komponenten jeweils um das System oder eine Einheit handelt, und ob es bei letzteren SW oder HW ist (vgl. Produktinhalte des Themas Dekomposition des Systems in Abbildung 6.24).

Der Ansatz in [110] verwendet Stereotypen, um die Inhalte eines Ergebnisses zum Vorgehensmodell in Bezug zu setzen. Abbildung 6.40 zeigt, wie das für das Beispiel aus Abbildung 6.39 konkret aussehen kann. Die beiden Einheiten werden hier hinsichtlich SW oder HW qualifiziert.

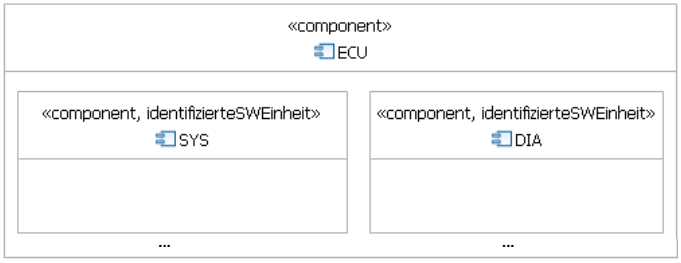


Abbildung 6.40 Auszeichnung von Inhalten für automatisierte Auswertung

Die Verwendung von Stereotypen ist jedoch nur eine konkrete Lösung für UML. Allgemein betrachtet handelt es sich dabei um einen Auszeichnungsmechanismus der das zu Grunde liegende Datenformat nicht verändert (vgl. lightweight language customization [111]). Nicht alle Datenformate bieten einen solchen Mechanismus. Entsprechende Auszeichnungen müssen dann ggf. virtuell (vgl. Kapitel 2.7) abgelegt werden müssen. Das Wiederum birgt

Risiken, dass die virtuell abgelegten Auszeichnungen bei Modifikationen verloren gehen, da sie vom Werkzeug als solche nicht erkannt und vor Verlust geschützt werden. Entsprechende Lösungen müssen daher je Datenformat erarbeitet werden. Für OpenDocument und GanttProject wurde dies beispielsweise in [112] betrachtet.

6.3.3 Steuerung

Bisheriges Vorgehen. Abbildung 6.41 zeigt, wie derzeit die Anwendung der Prozessbeschreibungssprache VM³ (V-Modell XT 1.3 Metamodell) zur Definition von Vorgehensmodellen durch Werkzeuge unterstützt erfolgt. Die Syntax von VM³ ist technisch ein XML Schema. Die damit beschriebenen Vorgehensmodelle sind folglich XML-Dateien. Zur Bearbeitung wird der V-Modell XT Editor eingesetzt, der im Wesentlichen ein XML-Editor ist.

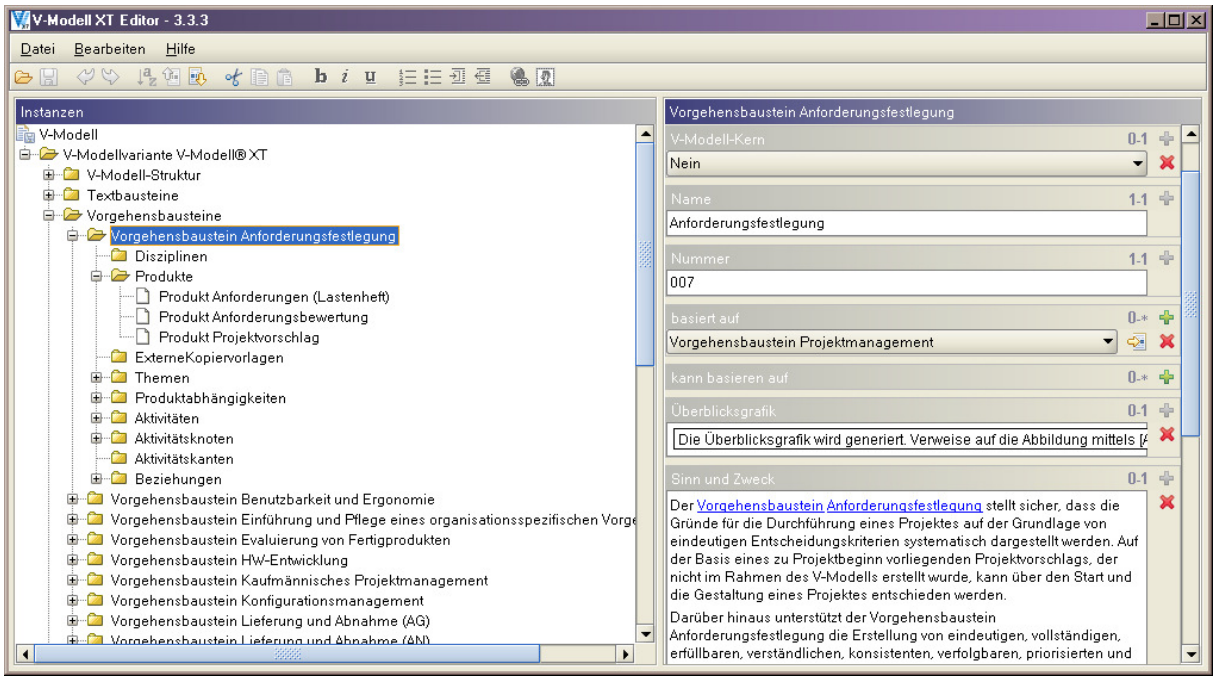


Abbildung 6.41 Vorgehensmodelldefinition mit dem V-Modell XT Editor

Im Unterschied zu einem normalen XML-Editor verfügt der V-Modell XT Editor jedoch einige Bedienkonzepte, die das Definieren von Vorgehensmodellen erleichtern[49].

- Im Strukturbaum (Abbildung 6.41, linker Bereich) werden nicht alle Hierarchien der XML-Datei dargestellt. Einige werden stattdessen als „logische“ Attribute behandelt, zu deren Bearbeitung spezifische Eingabeelemente (Abbildung 6.41, rechter Bereich) angeboten werden. Beispielsweise werden Links zwischen Elementen des Vorgehensmodells durch Dropdown-Felder angeboten.
- Im Strukturbaum wird für jeden Knoten ein Kontextmenü angeboten, durch das neue, untergeordnete Elemente eingefügt werden können. Zur Auswahl stehen jedoch nur solche, die an der entsprechenden Stelle passend zum betrachteten XML Schema sind.
- Es wird eine Exportfunktion zur Generierung diverser Formate angeboten – bspw. PDF [113] für Ausdrucke oder HTML [114] für eine Online-Präsenz. Um hierfür Bilder und eine Navigation zwischen den Inhalten zu ermöglichen, können die Beschreibungstexte auch HTML enthalten, welches bei der Generierung entsprechend verarbeitet wird.

[49] Gleichwohl sind die Bedienkonzepte nicht auf Vorgehensmodelle begrenzt.

Integration der Lösung in bisheriges Vorgehen. Die Integration der Lösung dreht sich bezüglich der Steuerung um die Frage, wie Gefügeketten bequem eingegeben werden können. Würde beispielsweise die Syntax von PBS$^+$ (Kapitel 6.2.1, Abbildung 6.2) naiv in ein XML Schema umgewandelt und dem V-Modell XT Editor als Grundlage für die Definition von Vorgehensmodellen gegeben, wären Gefügeketten nur schwer einzugeben: Zunächst wäre das Kontextmenü zum Anlegen neuer Gefüge sehr lang (und müsste ggf. gescrollt werden). Bei jedem angelegten Gefüge müsste manuell noch die Verbindung zum nächsten gezogen werden (falls vorhanden), was das Durchsuchen einer stetig wachsenden Dropdown-Liste der bisherig angelegten Gefüge nach sich zieht. Für die Praxis wäre das unzumutbar.

Eine naheliegende Auflösung dieses Problems bestünde darin, die Gefügeketten gemäß der UML-Aktivitätsdiagramme (Abbildung 6.3 - Abbildung 6.10) zu hierarchisieren. Die Gefüge einer Kette wären damit im Strukturbaum (vgl. Abbildung 6.41, links) nicht nebeneinander, sondern hierarchisch untereinander, mit einem Gefüge pro Ebene. Dies hilft bei der Eingabe, da so einerseits die Nachfolgerbeziehung der Gefüge durch die Hierarchie abgelöst wird, und andererseits im Kontextmenü die Auswahl auf die jeweils als nächstes gültig anlegbaren Gefüge reduziert wird. Allerdings leidet sowohl die Lesbarkeit einer solchen Gefügekette als auch die Möglichkeit, diese „in der Mitte“ zu bearbeiten.

Ein für die Praxis tauglicher Weg zur Eingabe von Gefügeketten basiert auf einer Texteingabe in Verbindung mit Autovervollständigung (vgl. [115]) und Code-Faltung (vgl. [116]). Dieser Weg gleicht im Grunde dem für die Eingabe von Code gängiger Programmiersprachen (wie Java) in gängigen Entwicklungsumgebungen (wie Eclipse). Bedingt durch die viel „engere“ Syntax der Gefügeketten haben hier jene beiden Konzepte jedoch eine qualitativ höhere Wirkung: Wie Abbildung 6.42 zeigt, kann bei Gefügeketten die Autovervollständigung nicht nur das aktuelle Terminal[50] vervollständigen, sondern auch solche, die zwingend im Anschluss oder sogar ganz am Ende der Gefügeketten folgen müssen[51]. Solche Terminale werden in Abbildung 6.42 grau dargestellt.

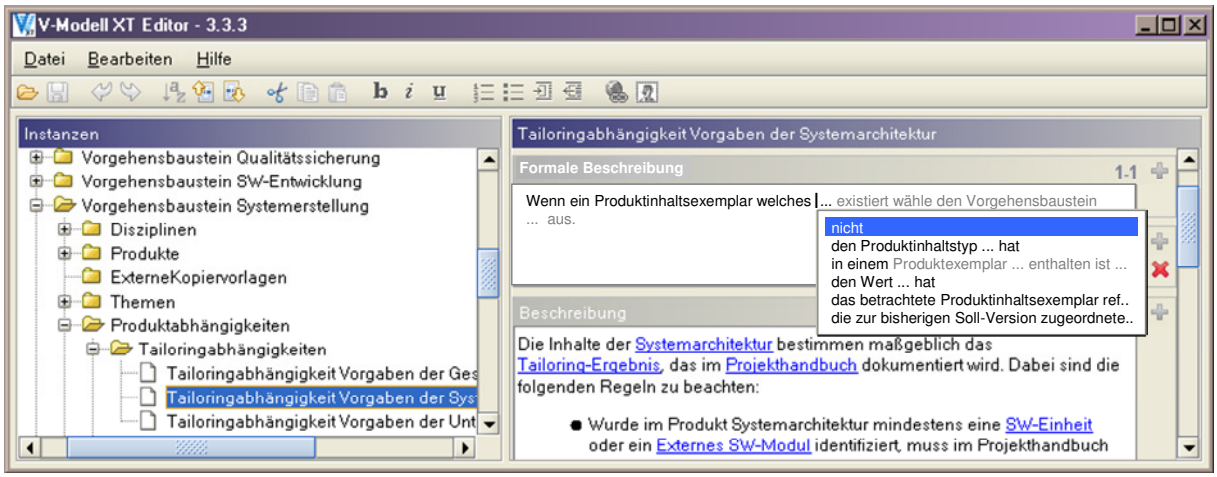


Abbildung 6.42 Autovervollständigung für Gefügeketten

Da Terminale einer Gefügeketten als Text hintereinander geschrieben werden, ist eine Regelung zum Umgang mit Detaillierungen und Bezügen zu Elementen des Vorgehensmodells nötig. Dazu werden erstere in Klammern gesetzt und letztere in die jeweiligen Terminale eingesetzt (vgl. Abbildung 6.43 oben).

[50] Terminal im Sinne der Abbildung Abbildung 5.40 in Kapitel 5.4.3.1. Bei Programmiersprachen umfasst ein Terminal dagegen meist nur ein Zeichen.
[51] Dieser Effekt kann bei Programmiersprachen nicht erreicht werden, da die Varianz des jeweils als nächsten möglichen zu groß ist.

Bevor die besondere Wirkung der Code-Faltung aufgezeigt wird, sei mit Abbildung 6.43 zunächst der Grundsätzliche Ansatz gezeigt, mit dem Gefügeketten unter Verwendung dieses Konzepts eingegeben und bearbeitet werden können.

Abbildung 6.43 Code-Faltung für Gefügeketten

In Programmiersprachen wird die Code-Faltung in der Regel bei Code-Blöcken angesetzt. So kann beispielsweise die Implementierung einer Funktion oder das Innere einer While-Schleife durch eine Faltung ausgeblendet werden. Für die an die natürliche Sprache angelehnten Gefügeketten dienen Klammern als Pendant (vgl. Abbildung 6.43, Mitte und unten). Als Stelle, an der eine Faltung vorgenommen wird, eignet sich die Detaillierungspunkt, die sich aus den Aufrufstellen zwischen den UML-Aktivitätsdiagrammen ergeben (vgl. Kapitel 5.4.3.1).

Phrasen können dabei wie normale Terminale abgebildet werden, wobei deren Auflösung über einen Tooltip angeboten wird (Abbildung 6.44).

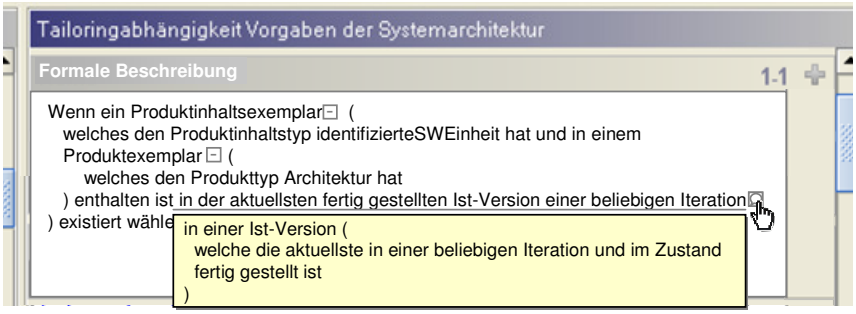


Abbildung 6.44 Darstellung von Phrasen für Gefügeketten

Der qualitative Vorteil der Code-Faltung für Gefügeketten besteht darin, dass auf Grund der „engen“ Syntax das Ausblenden mit einer „Projektion“ verbunden werden kann. Unter

Projektion sei hier der Effekt verstanden, dass bestimmte Teile des Ausgeblendeten das noch sichtbare überblenden. Abbildung 6.45 verdeutlicht dies an einem konkreten Beispiel: Durch das Ausblenden der Gefügekette „welches den Produkttyp Architektur hat“, überblendet der Teil „Architektur“ das verbleibende Terminal „Produktexemplar“. Somit hat ein Leser einen entscheidenden Hinweis über das Ausgeblendete, ohne dies überhaupt sehen zu müssen. Bei der zweiten Ausblendung (Abbildung 6.45, unten) überblendet der Teil „identifizierteSWEinheit“ das noch verbleibende Terminal „Produktinhaltsexemplar“.

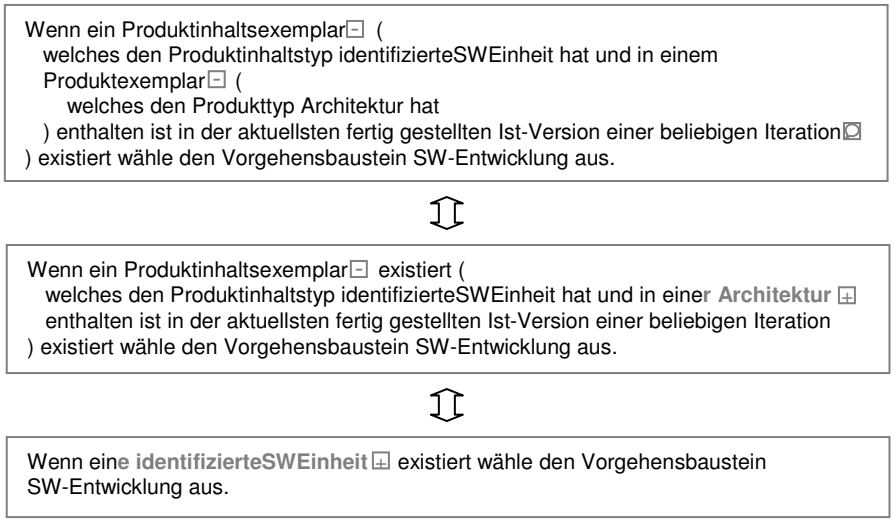


Abbildung 6.45 Code-Faltung und Projektion

Als Projektion kann prinzipiell eine beliebige Funktion verwendet werden, die auch unabhängig von der betrachteten Prozessbeschreibungssprache sein können. Allerdings erlaubt deren Berücksichtigung die Definition bessere Projektionen.

7 Zusammenfassung und Ausblick

Motivation und Zielsetzung. Bei Projekten sind bisherige Erfahrungswerte nur bedingt auf die neue Situation übertragbar. Doch je detaillierter ein Projekt geplant und dessen Fortschritt kontrolliert wird, desto eher kann Unvorhergesehenes erkannt und der weitere Ablauf entsprechend gesteuert werden. Dieser Regelkreis wird als Projektcontrolling bezeichnet. Schnell ist dabei die Grenze des manuell vertretbaren erreicht, so dass dessen Operationalisierung nützlich wird. Bisherige Ansätze beruhen jedoch auf für die Praxis ungeeigneten Voraussetzungen: Entweder müssen die Projektergebnisse in einer vordefinierten Reihenfolge erarbeitet, die Aktualisierung ausprogrammiert, oder die Vorgaben durch Graphersetzungen beschrieben werden. Entsprechend ist das Ziel dieser Arbeit, ein geeigneteres Verfahren für die Operationalisierung des Projektcontrollings aufzustellen.

Graphbasierte formale Sprachen als Ausgangspunkt. Eine formale Sprache wird als eine Menge von Graphen definiert, ein einzelnes Wort einer solchen Sprache ist somit ein konkreter Graph, der wiederum aus Knoten und Kanten besteht. Da eine Sprache in der Regel unendlich viele Wörter umfassen kann, ist, um darüber sprechen zu können, eine endliche Repräsentation notwendig die als Syntax bezeichnet wird. Eine Syntax kann dabei auf zwei verschiedene Weisen definiert werden: Entweder konstruktiv durch Produktionsregeln die eine Graphgrammatik bilden, oder analytisch durch ein Modell. Als Semantik einer Sprache wird eine Funktion bezeichnet die jedes Wort der Sprache auf einen beliebigen Wertebereich abbildet. Zur Definition solcher Funktionen werden für graphbasierte Sprachen Transformationen definiert, die wie Graphgrammatiken aus Produktionsregeln bestehen. Eine Produktionsregel besteht wiederum aus einem Such- und Ersetzungsmuster. Bei der Anwendung einer Produktionsregel auf einen Graph wird darin ein Ausschnitt gesucht der zum Suchmuster passt, und bei einem Treffer gemäß dem Ersetzungsmuster ausgetauscht. Durch Virtualisierung können Produktionsregeln nicht nur über Graphen, sondern auch über Produktionsregeln operieren.

Inkrementelle Transformation. Die inkrementelle Transformation ermöglicht es, einen bereits transformierten Graphen so erneut zu transformieren, dass das ursprüngliche Transformationsergebnis nur aktualisiert, und nicht von Grund auf neu erstellt wird. Zwischenzeitlich vorgenommene Änderungen können so erhalten werden. Erreicht wird dies durch das Führen einer Transformationshistorie, in dem jede Anwendung einer Produktionsregel hinsichtlich der vorgenommenen Ersetzungen protokolliert ist. Damit können frühere Anwendungen auch dann wiederholt werden, wenn der zu transformierende Graph geändert wurde.

Problem- und Aufgabenstellung. Der Problemgegenstand ist ein sich über die Zeit veränderndes Projekt. Es wird durch eine Produktbibliothek beschrieben, die sowohl den aktuellsten Projektstand, als auch die gesamte Historie des Projekts umfasst. Ein Projekt besteht dabei aus einer Menge von Ergebnissen (z.B. Dokumenten) und diese wiederum aus Inhalten. Durch Änderung der Inhalte eines Ergebnisses entstehen Ergebnis-Versionen. Durch Änderung der Ergebnisse (sowohl durch neue Versionen als auch durch weitere neue Ergebnisse) entstehen Projektstände. Bestimmte Ergebnisse und deren Inhalte dienen zur Anwendung des Projektcontrollings: Für die Planung definieren Soll-Projektstände die zu erarbeitenden Soll-Ergebnisse, für die Kontrolle geben Zuordnungen den erreichten Fortschritt an, und für die Steuerung definieren Vorgehensmodellelemente die anzuwendenden Vorgaben. Die Aufgabenstellung besteht darin, diese Vorgaben zu formalisieren, und damit die Schritte Planung, Kontrolle und Steuerung durch (Teil-)Automatisierungen zu unterstützen.

Lösung. Die Lösung integriert die inkrementellen Transformation mit Prozessbeschreibungssprachen. Den Kern bildet das Automatisierungskonzept, welches zeigt, wie die inkrementelle Transformation auf eine Produktbibliothek anzuwenden ist: Produktionsregeln werden durch Virtualisierung als Teil der Produktbibliothek gehalten, um selbst im Rahmen eines Steuerungsschrittes Gegenstand von Veränderungen sein zu können. Die inkrementelle Transformation wird induktiv durchgeführt, d.h. das Ergebnis einer Transformationsanwendung (der neue Stand der Produktbibliothek) wird als die Eingabe für die jeweils nächste verwendet. Die dafür notwendigen Grundlagen sind entsprechend erarbeitet worden. Die optimistische Planung zeigt, wie das Soll nicht nur für den jeweils nächsten Schritt, sondern auch für die zwingend danach folgenden, aufgestellt werden kann. Weiterhin wird gezeigt, dass die Kontrolle – der Abgleich zwischen Soll und Ist – auch dann noch als eine inkrementelle Transformation und auf einfache Weise gehandhabt werden kann, wenn dabei manuelle Entscheidungen zu treffen sind: Zuordnungen werden durch eine vordefinierte Produktionsregel erzeugt. Welche Zuordnungen jedoch tatsächlich zustande kommen, wird durch eine angepasste Redexauswahlfunktion entschieden. Diese bezieht den Projektleiter mit ein, der aus den möglichen Anwendungsstellen manuell solche heraussucht, die in seinem Sinne sind. Durch die inkrementelle Transformation sind für getroffene Entscheidungen sowohl deren Umsetzung, als auch deren Revidierung definiert. Schließlich erlaubt eine zusätzliche Erweiterung der inkrementellen Transformation um OCL, mit Produktionsregeln auf (hierarchische) Projektphasen leichter Bezug nehmen zu können.

Um die Lösung nicht auf eine konkrete Prozessbeschreibungssprache zu beschränken, wird eine Integrationsmethodik aufgestellt. Mit dieser kann eine Prozessbeschreibungssprache mit dem Automatisierungskonzept integriert werden. Dabei wird das Aufstellen geeigneter Sprachkonstrukte – und deren Unterlegung mit einer inkrementell ausführbaren Semantik – als ein Entwicklungsprojekt betrachtet: Zunächst werden Anforderungen an die Sprache gesammelt, um sie nur gerade so mächtig wie nötig, und damit so einfach wie möglich, zu halten. Aus diesen Anforderungen werden mit UML-Aktivitätsdiagrammen die mit der Sprache bildbaren Aussagen entworfen. Dabei wird sich an der natürlichen Sprache orientiert, um die Bedeutung der Aussagen intuitiv zu halten. Mit der in dieser Arbeit angegebenen Transformation können aus den UML-Aktivitätsdiagrammen neue Konstrukte für die zu integrierende Prozessbeschreibungssprache erhalten werden. In der Implementierung werden jene Konstrukte mit einer Semantik unterlegt. Dazu wird eine Transformation erstellt, die Instanzen dieser Konstrukte auf Produktionsregeln für das Automatisierungskonzept abbildet. Die Aktivitätsdiagramme dienen gleichzeitig aber auch zur Anwendung der neuen Konstrukte.

Evaluation. Zur Erprobung der Integrationsmethodik wurde die Prozessbeschreibungssprache V-Modell XT Metamodell in der Version 1.3 (kurz VM³) verwendet. Einige der in dieser Prozessbeschreibungssprache vorhanden Sprachkonstrukte konnten allein durch Unterlegung einer Abbildung auf die in der Integrationsmethodik bereitgestellten Strukturen und Produktionsregeln operationalisiert werden. Konkret betrifft dies das gesamte statische Tailoring sowie die Einplanung initialer Produkte. Für andere Vorgaben, wie das dynamische Tailoring, die erzeugenden Abhängigkeiten sowie das Bearbeitungszustandsmodell musste VM³ zunächst um passende Konstrukte erweitert werden, bevor auch hier eine Abbildung auf durch die Integrationsmethodik bereitgestellten Strukturen und Produktionsregeln erfolgte. Zur Erprobung der aufgestellten Konstrukte wurde ein konkretes Vorgehensmodell (ein Auszug aus dem V-Modell XT 1.3) beschrieben. Zur Eruierung der praktischen Umsetzbarkeit in Form eines Werkzeugs wurde ein entsprechender Integrationsansatz aufgestellt.

Ausblick. Diese Arbeit betrachtet ein Projekt als eine parallele Überarbeitung von Ergebnissen. Dabei hat jedes Ergebnis genau eine aktuellste Version. Dieses Verständnis kann auf Varianten (vgl. [15]) ausgeweitet werden, so dass von einem Ergebnis auch mehrere Varianten parallel weiterentwickelt, und später ggf. wieder zusammengeführt werden können. Analoges gilt für die Projektebene: Die Arbeit betrachtet genau einen aktuellsten Projektstand, dessen Überarbeitung einer total geordneten Kette von Soll-Projektständen folgt. Dies kann um zueinander parallel ablaufende Projektteile ausgebaut werden.

Für die Definition von Vorgehensmodellen wurde in dieser Arbeit ein allgemeines Rahmenwerk betrachtet. So wurden bei den Elementen eines Vorgehensmodells zwischen Ergebnis- bzw. Inhaltstypen, dem Zustandsmodell sowie Planungs- und Steuerungsvorgaben unterschieden. Durch eine weitere Spezialisierung dieser Basis können zusätzliche Techniken in das in dieser Arbeit aufgestellte Konzept integriert werden. Beispielsweise Änderungsoperationen (vgl. [88]), mit denen die zeitliche Veränderung eines Vorgehensmodells in Kombination mit Varianten behandelt werden kann. Somit kann die Parallelisierung nicht nur innerhalb eines Projektes, sondern darüberhinaus auch auf Organisationsebene erreicht werden.

Glossar

- **Anwender:** Je nach Betrachtungskontext ein →Projektleiter, ein →Prozessingenieur, ein Mitglied des →Projektteams oder eine beliebig andere nicht näher spezifizierte Person.
- **Produktexemplar:** Im einfachsten Sinne ein Dokument; allgemein ein beliebiges Erzeugnis in einem Projekt, dessen Typ (der Produkttyp) durch das eingesetzte Vorgehensmodell definiert ist.
- **Produktinhaltsexemplar:** Im einfachsten Sinne der Inhalt eines Dokuments; allgemein eine identifizierbare Information dessen Typ (der Produktinhaltstyp) durch das eingesetzte Vorgehensmodell definiert ist.
- **Projektabschnitt:** Ein identifizierbarer Zeitraum in einem Projekt.
- **Projektleiter:** Die für die Durchführung eines Projektes verantwortliche Person.
- **Projektteam:** An einem Projekt mit Arbeitskraft beteiligte Personen.
- **Prozessingenieur:** Die für das Aufstellen eines Vorgehensmodells verantwortliche Person.
- **Vorgabe:** Eine konkrete Anweisung oder Bestimmung, nach der in einem Projekt der Ablauf gesteuert, Ergebnisse eingeplant oder Ergebnisse erstellt werden.

Abkürzungsverzeichnis

- **bspw.**: beispielsweise
- **bzw.**: beziehungsweise
- **etc.**: et cetera
- **gdw.**: genau dann, wenn
- **ggf.**: gegebenenfalls
- **u.a.**: unter andere(n/m)
- **z.B.**: zum Beispiel

Literaturverzeichnis

[1] Willy Herroelen. Project Scheduling - Theory and Practice. In: Production and Operations Management, 2005, p.413-432.

[2] Manfred Bundschuh, Carol Dekkers. The IT Measurement Compendium: Estimating and Benchmarking Success with Functional Size Measurement. Springer, 2008.

[3] Software Process Engineering Metamodel. http://www.omg.org/technology/documents/formal/spem.htm (zitiert 27.05.2010)

[4] IBM - Rational ClearQuest - Rational ClearQuest - Software, URL: http://www-01.ibm.com/software/awdtools/clearquest/ (zitiert 27.05.2010)

[5] IBM collaboration software - Lotus software. URL: http://www-01.ibm.com/software/lotus/ (zitiert 27.05.2010)

[6] Gregory Alan Bolcer, Richard Taylor. Advanced workflow management technologies. Software Process -- Improvement and Practice, 1998.

[7] Jean-denis Grèze, Gail E Kaiser, Gaurav S Kc. Survivor: An Approach for Adding Dependability to Legacy Workflow Systems.

[8] Ansgar Schleicher. Formalizing UML-Based Process Models Using Graph Transformations. In: Applications of Graph Transformations with Industrial Relevance, 2000, p.141-145.

[9] CVS - Open Source Version Control. URL: http://www.nongnu.org/cvs/ (zitiert 27.05.2010)

[10] Apache Subversion. URL: http://subversion.apache.org/ (zitiert 27.05.2010)

[11] Walter F Tichy, Walter F Tichy. RCS - A System for Version Control. software - practice and experience, 1985, p.637-654.

[12] Yuehua Lin, Jing Zhang, Jeff Gray. Model comparison: A key challenge for transformation testing and version control in model driven software development. control in model driven software development. oopsla/gpce: best practices for model-driven software development, 2004, p.219-236.

[13] Maik Schmidt, Tilman Gloetzner. Constructing difference tools for models using the SiDiff framework. In: Companion of the 30th international conference on Software engineering. Leipzig, Germany, ACM, 2008, p.947-948.

[14] Udo Kelter, Jürgen Wehren, Jörg Niere. A Generic Difference Algorithm for UML Models. In: Software Engineering. Essen, 2005, p.105-116.

[15] Christian Bartelt. An Optimistic Three-way Merge Based on a Meta-Model Independent Modularization of Models to Support Concurrent Evolution. In: MODSE '08: Proceedings of the 2nd Workshop on Model-Driven Software Evolution. Athen, Griechenland, IEEE, 2008.

[16] Marcus Alanen, Ivan Porres. Difference and Union of Models. In: UML 2003 - The Unified Modeling Language: Modeling Languages and Applications. Springer-Verlag, 2003, p.2-17.

[17] Holger Giese, Robert Wagner. Incremental Model Synchronization with Triple Graph Grammars. proc. of the 9th international conference on model driven engineering languages and systems, 2006, p.543-557.

[18] David Hearnden, Michael Lawley, Kerry Raymond. Incremental Model Transformation for the Evolution of Model-Driven Systems. In: Model Driven Engineering Languages and Systems, 2006, p.321-335.

[19] Dirk Jäger, Ansgar Schleicher, Bernhard Westfechtel. AHEAD: A Graph-Based System for Modeling and Managing Development Processes. In Nagl et al., 1999, p.325-339.

[20] Bernhard Westfechtel. A Graph-Based System for Managing Configurations of Engineering Design Documents, 1996.

[21] Andy Schürr, Andreas J. Winter, Albert Zündorf. Graph Grammar Engineering with PROGRES. In: Proceedings of the 5th European Software Engineering Conference. Springer-Verlag, 1995, p.219-234.
[22] Markus Heller, Ansgar Schleicher, B. Westfechtel. A Management System for Evolving Development Processes, 2003.
[23] Duc-Hanh Dang, Martin Gogolla. On Integrating OCL and Triple Graph Grammars. In: Models in Software Engineering: Workshops and Symposia at MODELS 2008, Toulouse, France, September 28 - October 3, 2008. Reports and Revised Selected Papers, Springer-Verlag, 2009, p.124-137.
[24] Winfried Stier. Empirische Forschungsmethoden. Springer, 1999.
[25] Oliver Deiser. Einführung in die Mengenlehre : die Mengenlehre Georg Cantors und ihre Axiomatisierung durch Ernst Zermelo. Springer, 2004.
[26] Hans-Joachim Kowalsky, Gerhard O. Michler. Lineare Algebra. Walter de Gruyter, 2003.
[27] Ralf Hartmut Güting, Stefan Dieker. Datenstrukturen und Algorithmen. Vieweg+Teubner Verlag, 2004.
[28] Peter J. Pahl, Rudolf Damrath. Mathematische Grundlagen der Ingenieurinformatik. Springer, 2000.
[29] Gerhard Goos, Wolf Zimmermann. Vorlesungen über Informatik 1: Grundlagen und funktionales programmieren. Springer, 2005.
[30] Manfred Broy, Ralf Steinbrüggen. Modellbildung in der Informatik. Springer, 2003.
[31] Gerhard Goos. Vorlesungen über Informatik 3: Berechenbarkeit, formale sprachen, Spezifikationen. Springer, 1997.
[32] Oracle Corporation. Developer Resources for Java Technology, URL: http://java.sun.com/ (zitiert 27.05.2010)
[33] Uwe Schöning. Theoretische Informatik - kurzgefasst. Spektrum Akademischer Verlag, 1997.
[34] Peter Pepper, Petra Hofstedt. Funktionale Programmierung. Springer, 2006.
[35] Boris Reichel. Verteilte Auswertung attributierter Graphersetzungssysteme zur Verarbeitung massiver, graphartig strukturierter Daten. Herbert Utz Verlag, 1999.
[36] Henk Barendregt. The lambda calculus, its syntax and semantics, North-Holland, 1985.
[37] Steven Kelly, Juha-Pekka Tolvanen. Domain-specific modeling: enabling full code generation, IEEE, 2008.
[38] Sebastian Rönnau, Geraint Philipp, Uwe M. Borghoff. Efficient change control of XML documents. In: Proceedings of the 9th ACM symposium on Document engineering, Munich, Germany, ACM, 2009, p.3-12.
[39] Terrence W. Pratt. Pair grammars, graph languages and string-to-graph translations. Journal of Computer and System Sciences. 1971, p.560-595.
[40] Andy Schürr. Specification of Graph Translators with Triple Graph Grammars. In proc. Of the 20th int. Workshop on graph-theoretic concepts in computer science (wg `94), Herrsching
[41] Holger Giese, Robert Wagner. From model transformation to incremental bidirectional model synchronization. In Software and Systems Modeling, 2009, p.21-43.
[42] Alexander Königs. Model Transformation with Triple Graph Grammars. In model transformations in practice satellite workshop of models, 2005.
[43] Jordi Cabot, Robert Clarisó, Esther Guerra, Juan Lara. Analysing Graph Transformation Rules through OCL. In: Proceedings of the 1st international conference on Theory and Practice of Model Transformations. Zurich, Switzerland: Springer-Verlag, 2008, p.229-244.
[44] Benjamin C. Pierce. Types and programming languages. MIT Press, 2002.
[45] Benjamin C. Pierce. Advanced topics in types and programming languages. MIT Press,

2005.
[46] American National Standard for Information Systems — Coded Character Sets — 7-Bit American National Standard Code for Information Interchange (7-Bit ASCII), 1986.
[47] Object Management Group - UML. URL: http://www.uml.org/ (zitiert 27.05.2010)
[48] OMG. Object Constraint Language. URL: http://www.omg.org/technology/documents/formal/ocl.htm (zitiert 27.05.2010)
[49] Object Management Group. URL: http://www.omg.org/ (zitiert 27.05.2010)
[50] Gabriele Taentzer, Arend Rensink. Ensuring Structural Constraints in Graph-Based Models with Type Inheritance. In: Fundamental Approaches to Software Engineering. Springer-Verlag Berlin Heidelberg, 2005, p.64-79.
[51] Hartmut Ehrig, Ulrike Prange, Gabriele Taentzer. Fundamental Theory for Typed Attributed Graph Transformation. In: Graph Transformations. Springer-Verlag, 2004.
[52] Bernhard Westfechtel. Models and tools for managing development processes. Springer, 1999.
[53] Horst Reichel. Formal Specification of Data and Process Types. 2004.
[54] Thomas Kuehne, Daniel Schreiber. Can programming be liberated from the two-level style: multi-level programming with deepjava. SIGPLAN Not. 2007,42(10):229-244.
[55] DIN 69901 - Projektmanagement - Projektmanagementsysteme, 2009.
[56] Christian Bartelt, Edward Fischer, Thomas Ternité. Paradigmen zur Variabilitätsbeschreibung von Vorgehensmodellen. In: INFORMATIK 2009 – Im Focus das Leben. Bonn: Bonner Köllen, 2009, p. 3507–3521.
[57] Extensible Markup Language (XML) 1.0 (Fifth Edition). URL: http://www.w3.org/TR/2008/REC-xml-20081126/ (zitiert 27.05.2010)
[58] Jan Friedrich. Technische und semantische Transformation von Vorgehensmodellen, 2007.
[59] Hans-Jürgen Appelrath, Jochen Ludewig. Skriptum Informatik: Eine konventionelle Einführung. Vieweg+Teubner Verlag, 2000.
[60] William F. Clocksin, Christopher S. Mellish. Programming in Prolog. Springer, 2003.
[61] Helmut Herold, Michael Klar, Susanne Klar. Go to Objektorientierung. Pearson Education, 2001.
[62] Bjarne Stroustrup. Die C++ Programmiersprache. Pearson Education, 2000.
[63] Stanley M. Sutton, Leon J. Osterweil. The design of a next-generation process language. SIGSOFT Softw. Eng. Notes, 1997, p.142-158.
[64] LASER Process Working Group. Little-JIL 1.5 Language Report. Laboratory for Advanced Software Engineering Research, University of Massachusetts, 2006.
[65] Stanley M. Sutton, Dennis Heimbigner, Leon J. Osterweil. APPL/A: a language for software process programming. ACM Trans. Softw. Eng. Methodol, 1995, p.221-286.
[66] Wolfgang Emmerich, Volker Gruhn. FUNSOFT Nets: A Petri-Net based Software Process Modeling Language. In proceedings of the sixth international workshop on software specification and design. 1991, IEEE computer society, 1996, p.175-184.
[67] Reidar Conradi, Marianne Hagaseth, Jens-Otto Larsen, Minh Ngoc Nguyên, Bjørn P. Munch, Per H. Westby, u. a. EPOS: object-oriented cooperative process modelling. In: Software process modelling and technology. Research Studies Press Ltd., 1994, p.33-70.
[68] Naser Barghouti, Gail Kaiser. Scaling Up Rule-Based Software Development Environments. international journal on software engineering & knowledge engineering, 1992, p.59-78.
[69] G.E. Kaiser, N.S. Barghouti, M.H. Sokolsky. Preliminary experience with process modeling in the MARVEL software development environment kernel. In: Twenty-Third Annual Hawaii International Conference on System Sciences, p.131-140.
[70] G. Junkermann, B. Peuschel, W. Schäfer, S. Wolf. MERLIN: supporting cooperation in

software development through a knowledge-based environment. In: Software process modelling and technology. Research Studies Press Ltd., 1994, p.103-129.
[71] Burkhard Peuschel, Wilhelm Schäfer. Concepts and Implementation of a Rule-based Process Engine. in proceedings of the 14th international conference on software engineering, 1992, p.262-279.
[72] Christer Fernström. PROCESS WEAVER: Adding process support to UNIX. In: Proceedings of the Second International Conference on the Software Process, 1993, p. 12-26.
[73] S. M. J Sutton, B. S. Lerner, L. J. Osterweil. Experience Using the JIL Process Programming Language to Specify Design Processes. University of Massachusetts, 1997.
[74] The Workflow Management Coalition. Process Definition Interface - XML Process Definition Language.
[75] Worklfow Management Coalition. URL: http://www.wfmc.org/ (zitiert 27.05.2010)
[76] Edward Fischer. Workflowunterstützung für das Projektmanagement im V-Modell XT, 2005.
[77] OMG. BPMN 2.0. URL: http://www.omg.org/spec/BPMN/2.0/ (zitiert 27.05.2010)
[78] Philippe Kruchten. The rational unified process: an introduction. Addison-Wesley, 2004.
[79] HERMES — The HERMES Method. URL: http://www.hermes.admin.ch/ict_project_management/the-hermes-method (zitiert 27.05.2010)
[80] Unified Modeling Language, Version 2.2, Dokument: formal/2009-02-02. URL: http://www.omg.org/technology/documents/formal/uml.htm (zitiert 27.05.2010)
[81] V-Modell XT Bayern. URL: http://www.cio.bayern.de/internet/cio/4/19735/index.htm (zitiert 27.05.2010)
[82] BIT: V-Modell XT. URL: http://www.bit.bund.de/nn_490662/BIT/DE/Standards__Methoden/V-Modell_20XT/node.html?__nnn=true (zitiert 27.05.2010)
[83] Doris Rauh, Wolfgang Kranz. flyXT – Das neue Vorgehensmodell der EADS DE. In: Proceedings of the Conference on Software & Systems Engineering Essentials 2009 (SEE 2009). Technische Universität München, 2009.
[84] Alexander Bösl, Jan Ebell, Marco Kuhrmann, Andreas Rausch. Der Einsatz des V-Modell XT bei Witt Weiden: Nutzen und Kosten. SIGS Datacom. 2008, OBJEKTspektrum, p.31-37.
[85] Thomas Ternité, Marco Kuhrmann. Das V-Modell XT 1.3 Metamodell. München: Technische Universität München, 2009.
[86] IT-Beauftragte der Bundesregierung - V-Modell XT. URL: http://www.cio.bund.de/DE/IT-Methoden/V-Modell_XT/v-modell_xt_node.html (zitiert 27.05.2010)
[87] Tom Mens. A State-of-the-Art Survey on Software Merging. IEEE Trans. Softw. Eng. 2002, p.449-462.
[88] Thomas Ternité. Process Lines: A Product Line Approach Designed for Process Model Development. In: Software Engineering and Advanced Applications, Euromicro Conference. Los Alamitos, CA, USA, IEEE Computer Society, 2009. p.173-180.
[89] P. Baldan, A. Corradini, F. L. Dotti, L. Foss, F. Gadducci, L. Ribeiro. Towards a Notion of Transaction in Graph Rewriting. In Electronic Notes Theor. Comput. Science, 2008, p.39-50.
[90] Leila Ribeiro, Luciana Foss, Bruno Silva, Daltro Nunes. Model Transformation Using Graph Transactions. In: Proceedings of the 11th International Conference on Software Reuse: Formal Foundations of Reuse and Domain Engineering. Falls Church, Virginia:

Springer-Verlag, 2009, p.95-105.
[91] Andreas Rausch. Componentware: Methodik des evolutionären Architekturentwurfs, 2001.
[92] Duc-Hanh Dang. On Integrating Triple Graph Grammars and OCL for Model-Driven Development, 2009.
[93] David R. Brooks. C programming: the essentials for engineers and scientists. Springer, 1999.
[94] Larry E. Wood. User interface design: bridging the gap from user requirements to design. CRC Press, 1998.
[95] Elizabeth Hull, Ken Jackson, Jeremy Dick. Requirements engineering. Springer, 2005.
[96] Michael Deynet. Entwurf und Spezifikation einer Freisprechanlage nach V-Modell XT, 2005.
[97] Microsoft. Project Home Page - Microsoft Office Online. URL: http://office.microsoft.com/en-us/project/default.aspx (zitiert 27.05.2010)
[98] SAP. SAP - Business Management Software Solutions Applications and Services. URL: http://www.sap.com/index.epx (zitiert 27.05.2010)
[99] GanttProject Team. GanttProject. URL: http://www.ganttproject.biz/ (zitiert 27.05.2010)
[100] V-Modell XT Projektassistent, Version 1.3.1.
[101] The Eclipse Foundation. Eclipse. URL: http://www.eclipse.org/ (zitiert 27.05.2010)
[102] V-Modell-XT-Editor, Version 3.3.3. URL: http://sourceforge.net/projects/fourever/ (zitiert 27.05.2010)
[103] Peter Rinza. Projektmanagement: Planung, Überwachung und Steuerung von technischen und nichttechnischen Vorhaben. Springer, 1998.
[104] HERMES PowerUser. URL: http://ehermes.swissforge.net/tools/de/poweruser/ (zitiert 27.05.2010)
[105] Louis-Philippe Morency, Trevor Darrell. From conversational tooltips to grounded discourse: head poseTracking in interactive dialog systems. In: Proceedings of the 6th international conference on Multimodal interfaces. State College, PA, USA, ACM, 2004, p.32-37.
[106] IBM. IBM Software - Rational DOORS. URL: http://www-01.ibm.com/software/awdtools/doors/productline/ (zitiert 27.05.2010)
[107] Innovator. URL: www.mid.de/en/innovator/ (zitiert 27.05.2010)
[108] Microsoft. Visual C++ Tutorials, Library, and More on MSDN. URL: http://msdn.microsoft.com/en-us/visualc/default.aspx (zitiert 27.05.2010)
[109] ConTEXT Project. ConTEXT. URL: http://www.contexteditor.org/ (zitiert 27.05.2010)
[110] Christian Bartelt, Thomas Ternité, Matthias Zieger. Modellbasierte Entwicklung mit dem V-Modell XT. SIGS Datacom, 2005, OBJEKTspektrum, p.53-64.
[111] Marvin Zelkowitz. Advances in computers. Academic Press, 2007.
[112] Xibin Zhu. Werkzeugunterstütztes Controlling von Vorgehensmodellen, 2008
[113] Adobe - PDF Developer Center, PDF reference. URL: http://www.adobe.com/devnet/pdf/pdf_reference.html (zitiert 27.05.2010)
[114] HTML 4.01 Specification. URL: http://www.w3.org/TR/html401/ (zitiert 27.05.2010)
[115] Fabio Crestani, Paolo Ferragina, Mark Sanderson. String processing and information retrieval: 13th international symposium, SPIRE 2006, Glasgow, UK, October 11-13, 2006.
[116] J. Louise Finlayson. Providing a tailored overview of program source code. In: CHI '06 extended abstracts on Human factors in computing systems. Montréal, Québec, Canada, ACM, 2006, p.1743-1746.

Abbildungsverzeichnis

Definitionsverzeichnis